통보전달대면부(MPI)
프로그람작성지도서

국가과학원 콤퓨터과학연구소

우리식조작체계개발조

주체 98(2009)년 4월

차 례

머리말

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《과학기술을 빨리 발전시키기 위하여서는 또한 다른 나라의 선진과학기술을 적극 받아들여야 합니다.

다른 나라의 선진과학기술을 받아들이는것은 나라의 과학기술을 최단기간에 세계적수준으로 끌어올리기 위한 중요한 방도의 하나입니다.》
(《**김정일**선집》, 제 8 권, 249 페지)

현대과학기술의 급속한 발전은 콤퓨터 한대의 연산속도로서는 해결할수 없는 방대한 계산량을 가진 문제들을 수많이 제기하고있으며 이러한 대규모과학기술계산문제들을 신속정확히 해결하자면 높은 성능을 가진 병렬콤퓨터체계를 개발하고 그것을 효과적으로 리용하여야 한다. 이러한 현실적요구로부터 세계적으로 수백~수천개의 처리기들을 각이한 방식으로 결합한 성능높은 병렬콤퓨터체계들이 광범히 개발운영되고있다.

오늘날 병렬콤퓨터개발기술은 값비싼 전용콤퓨터로부터 개인용콤퓨터(PC)와 같은 일반화된 요소들을 고속이써네트망으로 결합한 무리방식의 병렬콤퓨터체계개발방향으로 나가고있다. 무리방식의 병렬콤퓨터는 전용병렬콤퓨터에 비하여 체계구축이 간단하며 확장성이 좋고 경제적효과성이 높은것으로 하여 현재 병렬콤퓨터체계개발의 기본추세로 되고있다. 그러나 그 우에서 실행되는 병렬응용프로그람의 개발은 프로그람작성자가 병렬프로쎄스들사이의 통신문제를 고려하여야 하므로 매우 복잡하고 어려운것으로 되고있다.

여러가지 병렬프로그람개발수단들의 특성을 잘알고 그것을 합리적으로 리용하는것은 방대한 계산량을 가진 계산문제들을 효과적으로 병렬처리하기 위한 필수적 요구로 나서고있다.

무리방식의 병렬콤퓨터에서 현재 가장 널리 리용되고있는 병렬프로그람작성수법은 통보전달대면부(Message Passing Interface)서고에 기초한 방법이다. 이 방법에서는 병렬프로그람을 통신함수들에 의하여 실행과정에 서로 통보를 주고받는 병렬프로쎄스들의 모임으로 고찰한다. 여기서 병렬프로그람작성은 보통 C나 포트란과 같은 언어로 작성된 직렬프로그람에 통신함수들을 리용하여 병렬성을 추가해주는 방법으로 진행한다. MPI병렬프로그람작성에 리용되는 통신함수들의 모임을 MPI서고라고 한다.

이 책은 3 개의 장으로 구성되여있다.

제 1 장에서는 병렬프로그람의 일반개념과 병렬프로그람개발의 한가지

수단인 MPI를 개발한 경위와 특징에 대하여 간단히 언급하고 MPI병렬 프로그람들을 개발할수 있는 기초적인 함수들과 그것들을 리용하는 실례들을 주었다.

제 2 장에서는 매개 MPI함수들의 기능을 통신방식과 마디점구성방법 등으로 나누어 실례를 들어주고있다.

제 3 장에서는 MPI프로그람작성방법을 실례를 들어 설명한다.

제 1 장 MPI의 기초

제 1 절 병렬프로그람과 MPI

1. 병렬프로그람의 개념

병렬프로그람이란 무엇인가를 알기 위하여 먼저 가장 간단한 병렬알고리듬을 고찰해보기로 한다.

아래에서는 사용자로부터 2 개의 옹근수값 n1 과 n2 를 입력받아 그 사이의 옹근수들을 모두 더한 합 n1+...+ n2 를 구하는 일감을 2 개의 처리기로 수행하는 병렬알고리듬을 보여주고있다. 2 개의 처리기에서 각각 실행되고있는 매개 프로쎄스는 아래와 같은 작업을 수행하게 된다.

단계	프로쎄스 1	프로쎄스 2
1	사용자로부터 n1, n2 를 입력받는다.	프로쎄스 1 로부터의 통보를 기다린다.
2	n1~n2 를 두개의 부분(n1~m, m+1~n2)으로 나눈다.	프로쎄스 1 로부터의 통보를 기다린다.
3	프로쎄스 2 에 통보(m+1~n2)를 보낸다.	프로쎄스 1 로부터의 통보(m+1~n2)를 받는다.
4	n1~m을 더한 결과(result1)를 구한다.	m+1~n2 을 더한 결과(result2)를 구한다.
5	m+1~n2 을 더한 결과(result2)를 프로쎄스 2 로부터 받는다.	프로쎄스 1 에 통보(result2)를 보낸다.
6	result1 과 result2 를 더한후 출력하고 끝낸다.	프로그람을 끝낸다.

매우 간단한 알고리듬이지만 프로그람에서는 망으로 련결하고 자료를 주고받는 등 일련의 처리절차를 서술하여야 한다.

이러한 통보전달방식의 병렬처리를 실현할 목적으로 통보전달함수들을 제공하였으며 이것을 서고화한것이 통보전달대면부(MPI)서고이다.

MPI병렬서고는 사용자에게 통보전달과 관련되는 다양한 함수들을 제공

해줌으로써 병렬응용프로그람작성자의 부담을 줄이고 보다 신뢰성있는 프로그람을 작성할수 있도록 한다.

MPI규약에 따르는 통보전달서고들가운데서 가장 널리 리용되고있는것은 LAM MPI와 MPICH를 들수 있다. 그러나 이것들은 모두 MPI표준규약에 따르므로 이것들사이에 원천코드가 100% 호환된다는 우점이 있다.

2. MPI의 특징

MPI는 Sillicon Graphics Origin 2000, Cray T3D, Cray T3E, IBM SP2 등과 같은 여러가지 기종의 고성능병렬콤퓨터체계들에서 통보전달방식의 병렬프로그람작성을 위한 기본수단으로 되고있다.

MPI는 동종 및 이종콤퓨터망에서도 리용할수 있으며 분산기억방식과 공유기억방식의 콤퓨터들에서 다 잘 동작한다.

MPI프로그람의 가장 중요한 특징은 MPI가 가상콤퓨터들사이의 가상통신망과 분산기억을 가진 가상다중콤퓨터를 제공하고있는것으로 하여 사용자가 병렬프로그람을 작성할 때 무리의 구조적특성을 고려하지 않아도 된다는것이다. 사용자는 다만 문제해결에 필요한 프로쎄스의 개수만을 요구하고 이 프로쎄스들사이의 위상만을 결정하면 된다. MPI는 사용자의 이러한 요구를 구체적인 병렬콤퓨터체계에서 실현한다. 결국 MPI는 병렬프로그람의 이식성을 보장하는 가상환경에서 동작한다.

3. MPI의 병렬화수법과 프로그람모형

병렬프로그람의 효과성은 계산시간 대 통신시간의 비로 나타난다. 총계산시간에서 통신시간이 작을수록 효과성은 더욱 높다. 통보전달방식의 병렬처리에서 계산과 통신사이의 최량비를 보장하는 방법은 호상작용이 적은 블로크사이의 병렬화를 진행하는 큰립도병렬화이다.

MPI병렬프로그람의 계산모형은 아래의 2가지로 나눈다.

MPMD계산모형: MPI프로그람은 프로그람 자체의 고유한 조종하에 자기기능을 수행하면서 통보들의 송수신을 위한 표준서고함수들에 의하여 호상작용하는 독자적인 프로쎄스들의 총체이다. 결국 MPI프로그람은 일반적으로 MPMD계산모형(Multiple Program-Multiple Data)을 실현한다.

SPMD계산모형: 여기서 모든 프로쎄스들은 같은 프로그람의 각이한 가지들을 실행한다. 이러한 방식은 과제가 같은 알고리듬에 의하여 부

분과제들로 충분히 분할될수 있는것과 관련된다. 현실에서는 이러한 프로그람작성모형과 자주 맞다들리게 된다. 이와 같은 SPMD(Single Program-Multiple Data)모형을 때로는 자료병렬화라고 한다. 간단히 말하여 이 수법에서는 과제의 초기자료들이 프로쎄스들에 따라 분할되지만 모든 프로쎄스들에서 알고리듬은 꼭같으며 이 알고리듬의 동작이 이 프로쎄스들에 있는 자료에 의하여 갈라진다.

제 2 절 MPI 프로그람작성의 초보

1. 가장 기본적인 6 개의 MPI 함수

MPI에는 200 여개 정도의 많은 함수들이 정의되여있다. 그러나 대부분의 MPI프로그람들은 아래에 소개되는 6 개의 기초적인 함수들의 조합으로 실현될수 있다.

이 함수들은 다음과 같다.

- MPI_Init(&argc, &argv)

프로그람의 입구파라메터들로 mpi환경을 초기화한다. 입구파라메터 지정이 없으면 기정값들을 리용한다. MPI_Init()함수는 모든 MPI프로그람들에서 다른 MPI함수들을 호출하기 전에 반드시 서술하여야 한다.

- MPI_Finalize()

mpi환경을 끝낸다.

MPI프로그람에서 호출되는 모든 MPI함수들은 MPI_Init()함수와 MPI_Finalize()함수사이에서 호출되여야 한다. MPI_Finalize()함수의 다음에 MPI함수들은 서술하는것은 아무런 의미도 없다.

- int MPI_Comm_rank(MPI_Comm comm, int *rank)

지정된 통신기(comm)안에서 자기(이 함수를 호출한 프로쎄스)의 프로쎄스번호(rank)를 얻는다.

- int MPI_Comm_size(MPI_Comm comm, int *size)

지정된 통신기(comm)에서 실행되는 프로쎄스의 개수(size)를 얻는다. size를 통신기의 크기라고도 부른다.

- int MPI_Send(void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

이 함수는 dest로 지정된 처리기에 통보를 보낸다.
매개 파라메터의 의미는 아래와 같다.
· message - 송신통보(자료)를 보관하고있는 완충기의 시작주소
· count - 송신완충기안의 요소개수
· datatype - 송신완충기요소의 자료형
· dest - 통보를 받아야 할 목적프로쎄스의 번호
· tag - 송신통보에 대한 표적
· comm -목적프로쎄스 dest와 자기 프로쎄스가 속해있는 통신기

- int MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

이 함수는 source로 지정해준 프로쎄스로부터 통보를 받는다.
· message - 수신통보를 보관할 완충기
· count - 수신될 통보의 요소개수(받는 요소개수보다 작으면 오유발생)
· datatype - 받을 통보의 자료형
· source - 송신프로쎄스의 번호
· tag - 받은 통보를 확인하기 위한 표적(MPI_Recv에서의 tag값과 MPI_Send에서의 tag값이 같아야 한다.)
· comm - source와 자기의 프로쎄스가 속해있는 통신기
· status - 통보수신상태에 대한 정보(source와 tag)

2. 실례프로그람

이 프로그람은 두 프로쎄스들사이에 MPI_Send와 MPI_Recv함수를 리용하여 자료를 주고 받는 과정을 보여주는 간단한 프로그람이다.
프로그람은 다음과 같다.

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
{
    int rank, size;
    char data[10];
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        strcpy(data, “process 0”);
       // MPI_CHAR는 C에서의 자료형 char와 같다.
       MPI_Send(data, 10, MPI_CHAR, 1, 123, MPI_COMM_WORLD);
    }
    elseif (rank == 1) {
    MPI_Recv(data, 10, MPI_CHAR, 0,123,MPI_COMM_WORLD,&status);
    printf( “massage=%s,        source=%d,        tag=%d\n”,        data,
   tatus.MPI_SOURCE, status.MPI_TAG);
    }
    MPI_Finalize();
    return 0;
}
```

3. 실례프로그람의 실행과 결과

우의 원천코드를 mpitest.c로 보관한 후 아래와 같은 일련의 과정을 거쳐서 실행해보자(MPI사용지도서를 참고할것). 먼저 mpicc지령으로 mpitest.c를 콤파일, 련결하여 실행가능한 MPI병렬프로그람 mpitest를 얻은 다음 mpirun지령을 리용하여 2개의 프로쎄스로 실행한다. 즉

```
[root~]# mpicc -o mpitest mpitest.c
[root~]# mpirun -np 2 mpitest
message=process 0, source=0, tag=123
[root~]#
```

제 3 절 집합통신

1. 기본적인 함수들

집합통신함수들은 통신기안에 속해있는 모든 프로세스들이 모두 함께 호출하여야 하는 함수들이다.

통신기란 프로쎄스그룹내에서 통신을 진행하기 위한 기구를 말한다.

MPI프로그람에서는 MPI_COMM_WORLD라는 기본통신기를 리용할수 있다. 이 통신기에는 생성된 모든 프로쎄스들이 포함된다. 그러나 사용자는 임의의 프로쎄스들로 구성되는 새로운 통신기들을 생성하고 리용할수도 있다(통신기와 관련된 MPI함수들(2 장 2 절)을 참고할것).

- int MPI_Bcast(void *message, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

통신기(comm)에 속한 모든 프로쎄스들에게 동일한 message를 전송한다. 여기서 root는 모든 프로쎄스들에게 통보를 방송하는 프로쎄스의 번호이다.

root프로쎄스의 message에만 송신할 자료가 들어있고 다른 프로쎄스의 message는 수신통보를 보관할 공간이다.

- int MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

모든 프로쎄스들에 다같이 들어있는 피연산수 operand가 op로 지정해준 연산을 수행하면서 집합되는데 그 집합연산결과는 프로쎄스 root의 result에 보관된다.

operand : 연산이 적용될 피연산수
result : 연산결과가 보관될 완충기(프로쎄스 root에서만 의미가 있음)
op :   수행될 연산을 지정하기 위한 연산코드
root : 연산결과가 보관될 프로쎄스번호

- int MPI_Barrier(MPI_Comm comm)

통신기(comm)에 속한 모든 프로쎄스들의 실행을 모두가 MPI_Barrier를 호출 할 때까지 차단시킴으로서 동기화를 진행한다.

- int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_comm comm)

모든 프로쎄스들이 프로쎄스 root에로 자료(sendbuf)를 보내며 root는 받은 자료를 root의 recvbuf에 송신프로쎄스의 번호순서대로 차례로 넣는다.

- int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

프로쎄스 root에 있는 sendbuf의 내용을 sendcount의 크기로 토막내여 매개 토막들을 모든 프로쎄스들에 있는 완충기 recvbuf에 전송한다. 이때 sendcount의 크기로 나눈 여러개의 토막들중 첫번째것은 첫번째 프로쎄스로, 두번째것은 두번째 프로쎄스로 보내는 식으로 송신한다.

MPI_Gather함수와 꼭 반대의 기능을 수행한다.

- int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_comm comm)

MPI_Gather함수와 마찬가지로 모든 프로쎄스의 완충기 sendbuf의 내용을 모든 프로쎄스의 완충기 recvbuf에 순서별로 모아놓는다.

MPI_Gather함수와의 차이점은 한개 프로쎄스에로만 수집되는것이 아니라 모든 프로쎄스에 다 수집된다는것이다.

- int MPI_Allreduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op, MPI_Comm comm)

MPI_Reduce와 마찬가지로 모든 프로쎄스의 피연산수 operand사이에 연산 op를 적용하지만 그 결과를 모든 프로쎄스의 result가 가진다는 점에서 차이난다.

2. 실례프로그람

아래에는 집합연산을 수행하는 MPI_Reduce함수의 사용실례를 보여준다.

```
#include <stdio.h>
#include "mpi.h"
```

```
main(int argc, char **argv)
{
    int rank, size, result;
    int data=10;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Reduce(&data, &result, 1, MPI_INT, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (rank == 0)
        printf("result = %d\n", result);
   MPI_Finalize();
   return 0;
 }
```

우의 프로그람을 mpireduce.c로 보관한 다음 콤파일, 련결, 실행한 결과는 아래와 같다.

```
[root~]# mpicc -o mpireduce mpireduce.c
[root~]# mpirun -np 3 mpireduce
result =30
[root~]#
```

제 4 절 MPI의 기타조작들

1. 통신자료의 그룹화

통신자료의 그룹화란 기억구역에 불련속적으로 놓여있는 여러개의 자료들을 모아서 한번에 전송할수 있도록 하나의 련속적인 자료로 만들어 주는것을 말한다. 이러한 기능들이 필요한것은 통보전달함수를 여러번 반복하여 통신하기보다는 이 자료들을 묶어서 하나의 자료형으로 만든 다음 한번의 통신함수리용으로 송신하는것이 더 편리하기때문이다.

통신자료의 그룹화조작을 위한 일부 함수들을 아래에 소개한다.

- int MPI_Type_struct(int count, int *array_of_block_lengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

이 함수는 C에서 새로운 구조체자료형을 선언하는것과 류사하게 새로운 MPI자료형을 선언한다.

실례로 C에서 struct test {int a[100]; float b[50]; double c;} *N; 이라는 새로운 구조체형을 선언한다고 할 때 그에 대응하는 MPI사용자정의자료형을 만든다고 하자. 이때 count는 새 자료형에 포함될 변수형(int, float, double)의 개수인 3, array_of_block_lengths는 포함될 블로크들의 길이인 (100, 50, 1), array_of_displacements는 새로운 자료형에서의 각 블로크들의 시작점 (어떤 기준으로부터의 변위)들의 배렬인 { &(N->a) - N, &(N->b) - N, &(N->c) -N } , array_of_types은 MPI자료형의 배렬 {MPI_INT, MPI_FLOAT, MPI_DOUBLE}, newtype은 사용자가 원하는 임의의 자료형이름(실례로 MPI_test)으로 해주면 된다.

- int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

불련속적인 기억기에 위치한 자료를 련속적인 기억기처럼 사용할수 있도록 한다.

- int MPI_Type_vector(int count, int block_length, int stride, MPI_Datatype element_type, MPI_Datatype *newtype)

규칙적인 길이와 간격을 가진 벡토르형태의 사용자자료형을 선언한다. 실례로 아래와 같이 불련속적인 기억구역에 배치되여있는 자료를 아래의 함수호출을 리용하여 하나의 련속적인 벡토르자료형으로 만든다.

MPI_Type_vector(3, 2, 2, MPI_INT, newtype)

1	2		3	4		5	6

- int MPI_Type_indexed(int count, int *array_of_block_lengths, int *array_of_displacements, MPI_Datatype element_type, MPI_Datatype *newtype)

불규칙적인 길이, 불규칙적인 간격을 가진 배렬을 새로운 형태로 선언한다.

array_of_block_lengths는 각 블로크의 길이의 배렬인 {2, 1, 3}

array_of_displacements는 각 블로크의 변위의 배렬 {0, 3, 6}

1	2		3			4	5	6

- MPI_Type_commit(MPI_Datatype *newtype)

MPI에서 선언해준 새로운 사용자정의자료형을 MPI에서 사용할수 있게 한다. MPI에서 선언한 새로운 자료형들(기본자료형이 아닌)은 형선언시에는 통신완충구역의 내용과 련결되지 않는다. 따라서 이 함수를 리용하여 체계로부터 사용허가를 반드시 받아야 한다. 그렇치 않으면 우에서 설명된 통신자료의 그룹화조작들에서 생성한 새로운 자료형들을 알수 없다는 오유통보를 내보낸다.

- int MPI_Pack(void *pack_data, int in_count, MPI_Datatype datatype, void *buffer, int size, int *position_ptr, MPI_Comm comm)

pack_data에 있는 count개의 자료를 송신에 앞서 buffer에로 묶기해준다.

position_ptr는 자료가 보관될 buffer위치의 지적자이며 묶기한 후에는 이 지시기가 자동적으로 증가한다.

size는 바이트단위로 계산된 buffer의 크기이다.

일부 다른 통신서고들(MPI가 아닌)은 불련속적인 몇개의 구역으로 이루어진 자료들을 전송하기 위해 자료묶기함수, 자료풀기함수들을 가지고 있다. 여기서는 자료들을 송신하기 전에 어떤 완충구역에 묶어놓고 수신후에는 이 완충구역에서부터 자료를 푼다. MPI에서 사용자정의자료형생성은 대부분의 경우에 자료의 명시적인 묶기와 풀기를 피할수 있게 해준다.

응용프로그람에서는 전송하고 수신할 자료의 위치만을 지정하며 MPI는 몇개의 불련속적인 구역으로 이루어진 완충기에 직접 접근한다.

MPI에서 묶기와 풀기조작들은 이전 서고와의 호환성을 보장한다. 또한 이것들은 MPI에서는 제공되지 않는 일부 기능들을 보장하고있다. 실례로 통보는 개별적인 부분에 의해 수신될수 있으며 이 부분에서는 후에 실행되는 수신조작이 이전 부분의 내용에 관계될수 있다. 또다른 좋은 측면은 묶기 및 풀기조작들이 보조적인 통신서고들을 쉽게 활용할수 있게 한다는

것이다.

- int MPI_Unpack(void *buffer, int size, int *position_ptr, void *unpack_data, int count, MPI_Datatype datatype, MPI_comm comm)

MPI_Pack조작으로 묶기한 자료를 풀어준다.

buffer는 묶기된 자료가 보관되어있는 기억구역이다.

position_ptr는 묶기된 자료 buffer의 첫 위치에 대한 지적자이며 풀기(unpack)한 후에는 지시기가 자동적으로 증가한다.

size는 바이트단위로 계산된 완충기의 크기이다.

count개의 자료를 풀어서 datatype형으로 unpack_data에 복사해준다.

2. 그룹과 위상(Topology)

다양한 통신기를 만들고 처리기들의 위상을 선언할수 있는 함수들로 구성된다.

- int MPI_Group_incl(MPI_Group old_group, int new_group_size, int *ranks_in_old_group, MPI_Group *new_group)

old_group에서 rank_in_old_group안의 번호들을 가진 new_group_size 개수의 프로세스들로 새로운 그룹인 new_group을 만든다.

- int MPI_Comm_create(MPI_Comm old_comm, MPI_Group new_group, MPI_Comm *new_comm)

그룹을 통신기로 만들어준다. MPI_Comm_group과 MPI_Group_incl 함수와는 달리 집합연산은 단독으로 수행되서는 안되며 반드시 모든 프로세스가 동시에 수행되여야 하는 조작이다.

- int MPI_Comm_split(MPI_Comm old_comm, int split_key, int rank_key, MPI_Comm *new_comm)

동일한 split_key를 가지고 호출하는 프로세스들을 묶어서 동일한 통신기로 만든다. rank_key는 프로세스번호를 결정할 때 작은 순서대로 작은 번호를 부여받는다.

프로세스 0, 1, 2 는 split_key 1 로 호출하고 프로세스 3, 4, 5 번은

split_key 2 로 호출했을 때 동일한 이름(new_comm)을 가진 두개의 통신기가 생성된다. 만약 0 번 프로쎄스에서 new_comm으로 MPI_Bcast를 호출하면 0, 1, 2 번으로 전송되고 3 번 프로쎄스에서 new_comm으로 MPI_Bcast를 호출하면 3, 4, 5 번으로 전송된다.

- int MPI_Cart_create(MPI_Comm old_comm, int number_of_dims, int *dim_sizes, int *periods, int reorder, MPI_Comm *cart_comm)

새로운 통신기 cart_comm을 생성해준다.

old_comm과 cart_comm에 속해있는 프로쎄스들은 같지만 그것들의 번호는 같지 않다.

cart_comm의 프로쎄스들은 number_of_dims차원의 행렬구조(dim_size *dim_size ...)로 재구성되며 그에 맞게 새로운 번호를 가지게 된다.

periods는 각 차원이 비선형인가 혹은 선형인가를 지정하는 기발이다.

reorder는 프로쎄스번호의 재순서화를 지시하는 기발이다.

- int MPI_Cart_rank(MPI_Comm comm, int *coordinates, int *rank)

coordinates[]의 정보를 가지고 그에 해당하는 프로쎄스의 번호를 구한다.

coordinates[]={2,4} 이면 프로쎄스배치행렬에서 2 행 4 렬의 프로쎄스 번호를 돌려준다.

coordinates[]는 몇렬 몇행의 정보를 가지고있다.

- int MPI_Cart_coords(MPI_Comm comm, int rank, int number_of_dims, int *coordinates)

rank를 가지고 그에 해당하는 coordinates[]를 구한다.

2 행 4 렬에 위치한 프로쎄스번호가 6 일 때 rank=6 로 함수를 호출하면 coordinates[]={2, 4} 를 돌려준다.

- int MPI_Cart_sub(MPI_Comm grid_comm, int *varying_coords, MPI_Comm *comm)

2 * 2 행렬인 경우 2 개의 통신기를 만든다.

MPI_Cart_sub(grid_comm, {0, 1}, comm)을 사용하여 매개 렬을 동일한 통신기로 묶어준다.

varying_coords[]는 매개 차원이 comm에 속하는가를 알려주는 론리값

이다.

3. MPI 자료형과 MPI 연산형

MPI프로그람에서 자료를 전송 또는 송신할 때는 반드시 자료형을 함께 지정해주어야 한다. 모든 자료통신은 바이트단위로 진행되기때문에 이것을 원래의 자료형으로 해석하기 위해서는 자료형을 반드시 지정해야 한다.

C와 포트란의 모든 기초자료형에 대응하는 MPI자료형은 아래와 같다.

MPI자료형	C 자료형	MPI자료형	포트란자료형
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_SHORT	signed short int	MPI_INTEGER	Integer
MPI_INT	signed int	MPI_REAL	Real
MPI_LONG	signed long int	MPI_DOUBLE_PRECISION	double precision
MPI_UNSIGNED_CHAR	unsigned char	MPI_COMPLEX	Complex
MPI_UNSIGNED_SHORT	unsigned short int	MPI_LOGICAL	Logical
MPI_UNSIGNED	unsigned int	MPI_BYTE	8 bit
MPI_UNSIGNED_LONG	unsigned long int	MPI_PACKED	MPI_Pack() 또는 MPI_Unpack() 조작으로 묶기 또는 풀기한 자료
MPI_FLOAT	float		
MPI_DOUBLE	double		
MPI_LONG_DOUBLE	long double		
MPI_BYTE	8 bit		
MPI_PACKED	MPI_Pack() 또는 MPI_Unpack() 조작으로 묶기 또는 풀기한 자료		

MPI_Reduce()와 같은 일부 집합연산함수들은 매개 프로쎄스로부터 어떤 연산을 수행하면서 자료를 모아서 결과를 돌려주게 된다.

이때 수행할 연산을 지정해주어야 하는데 MPI에서 기정으로 리용할수 있는 연산형들은 아래와 같다.

MPI집합연산형	의미	연산수자료형(C)	연산수자료형(포트란)
MPI_MAX	최대값(maxmum)	integer, float	integer, real, complex
MPI_MIN	최소값(minimum)	integer, float	integer, real, complex
MPI_SUM	총합(sum)	integer, float	integer, real, complex
MPI_PROD	곱하기(product)	integer, float	integer, real, complex
MPI_LAND	론리적(logical AND)	integer	logical
MPI_BAND	비트론리적(bit-wise AND)	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	론리합(logical OR)	integer	logical
MPI_BOR	비트론리합(bit-wise OR)	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	배타론리합(logical XOR)	integer	logical
MPI_BXOR	배타적인 비트론리합(bit-wise XOR)	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	최대값과 위치	float, double, long double	real, complex, double precision
MPI_MINLOC	최소값과 위치	float, double, long double	real, complex, double precision

제 5 절 MPI 프로그람시작

순차프로그람 《Hello world》와 같은 간단한 MPI프로그람 《I am process ID》를 작성하기로 한다.

```
#include   <stdio.h>
#include    “mpi.h”

main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);
    // size 에 함께 수행된 처리기의 개수가 할당된다.
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // rank에 자기(프로세스)의 번호가 할당된다.
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // 매개 프로세스는 아래의 printf문을 수행한다.
    printf("I am process %d\n", rank);

    MPI_Finalize();
    return 0;
}
```

아래의 지령은 3 개의 처리기가 동일한 프로그람을 수행하고있는것을 보여주고있다.

```
[root@node56 ~]# mpirun -np 3 test1
I am process 0
I am process 1
I am process 2
[root@node56 ~]#
```

이제 매개 프로세스를 어떻게 조종하는지 실례프로그람을 통해 알아보자.

```
#include   <stdio.h>
#include    “mpi.h”
```

```
main(int argc, char **argv)
 {
      int rank, size;

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      // 프로쎄스번호가 0 이면
      if(rank == 0)
        printf("%d processes is runned\n", size);
     // 프로쎄스번호가 0 이 아닌 프로쎄스이면
      else
        printf("I am process %d\n", rank);

      MPI_Finalize();
      return 0;
 }
```

```
[root~]# mpirun -np 3 test2
3 processes is runned
I am process 1
I am process 2
```

다음은 MPI병렬프로그람의 아주 간단한 몇가지 실례이다.

C프로그람실례	포트란 실례
Hello_world.c	Hello_world.f
sendrecv.c	sendrecv.f
matv.c	matv.f
wtime.c	

Hello_world.c

```
#include <stdio.h>
#include  “mpi.h”

main(int argc, char **argv)
{
   int node, numtask;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &node);
   MPI_Comm_size(MPI_COMM_WORLD, &numtask);

   printf("Hello World!!! from node = %d\n", node);

   MPI_Finalize();
   return 0;

}
```

sendrecv.c

```
#include <stdio.h>
#include  “mpi.h”

main(int argc, char** argv)
{
   int node, numtask, nextup, nextdn, itag, size;
   float sendbuf, recvbuf;
   MPI_Status stat;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &node);
   MPI_Comm_size(MPI_COMM_WORLD, &numtask);

   if (node != (numtask-1))      nextup = node + 1;
```

```
  else        nextup = 0;

  if (node != 0)      nextdn = node - 1;
  else     nextdn = numtask-1;
  printf("node = %d, nextup = %d, nextdn = %d\n", node, nextup,
nextdn);
  sendbuf = node*0.1;
  itag = 22;
  size = 1;

MPI_Send(&sendbuf, size, MPI_REAL, nextdn, itag,
            MPI_COMM_WORLD);
MPI_Recv(&recvbuf, size, MPI_REAL, nextup, itag,
            MPI_COMM_WORLD, &stat);

printf("from %d received %e\n", nextup, recvbuf);

MPI_Finalize();
return 0;

}
```

matv.c

```
#include <stdio.h>
#include  "mpi.h"

main(int argc, char** argv)
{
  int node, numtask, nextup, nextdn, itag, size, i, j;
  float sendbuf[10], recvbuf[10], c[10], a[10][10];

  MPI_Status stat;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &node);
  MPI_Comm_size(MPI_COMM_WORLD, &numtask);

  if (node != (numtask-1)) nextup = node + 1;
  else   nextup = 0;

  if (node != 0) nextdn = node - 1;
  else   nextdn = numtask-1;

  printf("node = %d, nextup = %d, nextdn = %d\n", node, nextup,
nextdn);

  for (i = 0; i < 10; i++) {
    c[i] = 0.0;
    sendbuf[i] = 1.0*i*(node+1);
    for (j = 0; j < 10; j++) a[i][j] = (i*1.0 + j*0.1)*(node+1.0);
  }

  itag = 22;
  size = 10;
  MPI_Send(&sendbuf, size, MPI_REAL, nextdn, itag,
           MPI_COMM_WORLD);
  MPI_Recv(&recvbuf, size, MPI_REAL, nextup, itag,
           MPI_COMM_WORLD, &stat);

  for (i = 0;i < 10;i++)
    for (j = 0;j < 10;j++) c[i] = c[i] + a[i][j]*recvbuf[j];

  for (i = 0;i < 10;i++) printf("%e \n", c[i]);

  MPI_Finalize();
  return 0;
}
```

wtime.c

```
#include <stdio.h>
#include <time.h>
#include  “mpi.h”

main(int argc, char **argv)
  {
      int rank;
      double start, end;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      start=MPI_Wtime();
      sleep(7);
      end=MPI_Wtime();

        printf("PROC=%d   Wall clock time=%f\n", rank, end-start);

        MPI_Finalize();
        return 0;
}
```

포트란 실례

Hello_world.f

```
program hello_mpi2

         character*80 name
         include    “mpif.h”

         call MPI_INIT(ierr)
```

```
        call MPI_GET_PROCESSOR_NAME(name,ilength,ierr)

        write(6,*) "Hello World!!! from node ", name

        call MPI_FINALIZE(ierr)

        stop
        end
```

send_recv.f

```
program comm_mpi

        include 'mpif.h'

        integer istatus(MPI_STATUS_SIZE)

        call MPI_INIT(ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD, node, ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, numtask, ierr)

        if (node .ne. (numtask-1)) then     nextup = node + 1
        else     nextup = 0
        endif

        if (node .ne. 0) then   nextdn = node - 1
        else     nextdn = numtask-1
        endif

        write(6,*) 'node = ', node,' ; ', 'nextup = ', nextup,
      #        'nextdn = ', nextdn

        sendbuf = node*0.1
```

```
      itag = 22

      call MPI_SEND(sendbuf, 1, MPI_REAL, nextdn, itag,
     #        MPI_COMM_WORLD, ierr)
      call MPI_RECV(recvbuf, 1, MPI_REAL, nextup, itag,
     #        MPI_COMM_WORLD, istatus, ierr)

      write(6,*) 'from ', nextup, ' receive ', recvbuf

      call MPI_FINALIZE(ierr)

      stop
      end
```

matv.f

```
program comm_mpi

      include 'mpif.h'

      integer istatus(MPI_STATUS_SIZE)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, node, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtask, ierr)

      if (node .ne. (numtask-1)) then   nextup = node + 1
      else     nextup = 0
      endif

      if (node .ne. 0) then   nextdn = node - 1
      else     nextdn = numtask-1
      endif
      write(6,*) 'node = ', node,' ; ', 'nextup = ', nextup,
```

```
#       'nextdn = ', nextdn

 sendbuf = node*0.1

 itag = 22

 call MPI_SEND(sendbuf, 1, MPI_REAL, nextdn, itag,
#       MPI_COMM_WORLD, ierr)
 call MPI_RECV(recvbuf, 1, MPI_REAL, nextup, itag,
#       MPI_COMM_WORLD, istatus, ierr)

 write(6,*) 'from ', nextup, ' receive ', recvbuf

 call MPI_FINALIZE(ierr)

 stop
 end
```

제 2 장 병렬프로그람작성서고 MPI

우에서는 일반적인 병렬프로그람과 MPI의 기초적인 개념들과 일부 실례에 대하여 보았다. 여기서는 앞에서 설명한 MPI의 기본함수들과 함께 다른 다양한 MPI의 함수들에 대하여 기능별로 실례를 주면서 고찰한다.

여기서는 C프로그람에서 리용되는 MPI함수들을 설명한다.

포트란프로그람에서 모든 MPI함수들(MPI_WTIME과 MPI_WTICK는 제외)은 C에서와는 달리 파라메터목록의 끝에 보충적으로 옹근수파라메터 ierr를 가진다. 이것은 C에서 MPI함수들의 귀환값과 같은 의미를 가진다.

포트란에서 MPI함수들은 부분루틴이며 call명령을 리용하여 호출할수 있다. 모든 MPI객체들(실례로 MPI_Datatype, MPI_Comm)은 포트란에서 INTEGER형이다.

제 1 절 그룹과 그룹연산

1. 그룹

일부 응용프로그람들에서는 프로쎄스들을 독립적인 작업을 하도록 프로쎄스그룹으로 분할하는것이 편리하다.

례를 들면 행렬계산에서 방송(하나를 전체에 전송)조작에 의하여 행렬의 대각선에 따르는 프로쎄스들에게만 통보를 전달할 필요가 있다면 이 대각선프로쎄스들을 하나의 그룹으로 묶어 이 그룹만이 통신조작에 참가하도록 하면 속도가 훨씬 더 빨라진다.

MPI프로쎄스들은 실행독립인 대상들이며 그룹은 프로쎄스식별자(번호)들의 모임이다. 그룹에서 매개 프로쎄스는 번호로 표시한다. 그룹에서 프로쎄스번호는 0 부터 시작하여 그 그룹에 속한 프로쎄스개수에서 1 을 덜어낸 수값까지의 순서붙은 옹근수이다. 그룹에 한하여 임의의 복잡도를 가진 프로쎄스들사이의 련관구역을 구축할수 있는 모임론적인 그룹연산들이 MPI에 정의되여있다.

기정그룹인 MPI_GROUP_EMPTY는 성원이 전혀 없는 빈 그룹이다.

2. 기본함수들

- MPI_Group_size

그룹의 크기를 되돌린다.

문법

```
#include "mpi.h"
int MPI_Group_size( MPI_Group group, int *size )
```

입구파라메터

group : 그룹(손잡이)

출구파라메터

size : 그룹내에 있는 프로세스개수

- MPI_Group_rank

주어진 그룹에서 자기 프로세스의 번호를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Group_rank( MPI_Group group, int *rank )
```

입구파라메터

group : 그룹(손잡이)

출구파라메터

rank : 그룹에서 호출한 프로세스의 번호(옹근수) 또는 이 프로세스가 성원이 아니면 상수값인 MPI_UNDEFINED

- MPI_Group_translate_ranks

서로 다른 그룹중 한개 그룹에서의 프로세스번호를 다른 그룹에서의 프로세스번호로 변환한다.

문법

```
#include "mpi.h"
int MPI_Group_translate_ranks( MPI_Group group_a, int n, int
```

*ranks_a, MPI_Group group_b, int *ranks_b)

입구파라메터

group_a	그룹 1(손잡이)
n	ranks_a와 ranks_b 배렬들에서 번호개수(옹근수)
ranks_a	group_a에서 0 또는 그 이상의 유효한 번호배렬
group_b	그룹 2(손잡이)

출구파라메터

ranks_b : group_b에서 대응하는 번호들의 배렬, 일치한 번호의 배렬이 존재하지 않을 때 MPI_UNDEFINED값이다.

[실례 1.1] group1 이 그룹 {a, b, c, d, e, f}의 이름이고 group2 가 그룹 {d, e, a, c}의 이름이라고 하자. ranks_a={0, 5, 0, 2}이라고 하자. 그러면 MPI_Group_translate_ranks를 호출하면 group2에 프로쎄스모임 {a, f, a, c}를 주며 ranks_b={2, ?, 2, 3}이 될것이다.

여기에서 ?은 MPI_UNDEFINED값이다.

- MPI_Group_compare

두 그룹을 비교한다.

문법

```
#include "mpi.h"
  int MPI_Group_compare(MPI_Group group1, MPI_Group group2,
                                  int *result )
```

입구파라메터

group1	그룹 1(손잡이)
group2	그룹 2(손잡이)

출구파라메터

result: 두 그룹의 순서와 성원들이 같으면 MPI_IDENT인 옹근수, 성원들만 같으면 MPI_SIMILAR, 그 외에는 MPI_UNEQUAL인 옹근수값

3. 그룹구축자와 그룹연산

여러가지 그룹연산들을 리용하여 이미 존재하는 그룹들로부터 새로운 그룹을 구축할수 있으며 이때 그룹구축자를 리용한다. MPI는 시작초기부터 그룹을 구축하도록 하는 기능을 제공하지 않으며 구축은 다만 미리 정의되여있는 그룹으로부터만 가능하다. 이 기정그룹은 기정통신기 MPI_COMM_WORLD로부터 만들수 있다.

- MPI_Group_intersection

현존 두개 그룹의 사귐부분으로 새 그룹으로 창조한다.

문법

```
#include "mpi.h"
int MPI_Group_intersection( MPI_Group group1, MPI_Group
                            group2, MPI_Group *newgroup )
```

입구파라메터

group1 그룹 1(손잡이)

group2 그룹 2(손잡이)

출구파라메터

newgroup : 공통부분그룹(손잡이)

- MPI_Group_difference

두개 그룹의 차로부터 새 그룹을 만든다.

문법

```
#include "mpi.h"
int MPI_Group_difference( MPI_Group group1, MPI_Group
                          group2,  MPI_Group *newgroup )
```

입구파라메터

group1 첫째 그룹(손잡이)

group2 둘째 그룹(손잡이)

출구파라메터

newgroup : 새 그룹(손잡이)

- MPI_Group_union

두 그룹을 결합하여 새 그룹을 창조한다.

문법

```
#include "mpi.h"
int MPI_Group_union( MPI_Group group1, MPI_Group
                     group2, MPI_Group *newgroup )
```

입구파라메터

group1 첫째 그룹(손잡이)

group2 둘째 그룹(손잡이)

출구파라메터

newgroup : 합쳐진 그룹(손잡이)

[실례 1.2.] group1={a, b, c, d} 이고 group2={d, a, e} 일때

group1 ∪ group2={a, b, c, d, e} (합)

group1 ∩ group2={a, d} (적)

group1 \ group2 = {b, c} (차)

- MPI_Group_excl

현재 그룹에서 지정된 성원들을 배제하고 재순서화하여 새로운 그룹을 창조한다.

문법

```
#include "mpi.h"
int MPI_Group_excl( MPI_Group group, int n, int *ranks,
                    MPI_Group *newgroup )
```

입구파라메터

group 그룹

n　　　ranks배열에서의 원소개수(옹근수)

ranks　newgroup에 들어가지 않는 group에서의 번호들의 배렬

출구파라메터

newgroup: 생성된 새로운 그룹, 그룹 group에서의 순서를 보존한다.

[실례 1.3] group= {a, b, c, d, e, f} 이고 ranks = {3, 1, 2} 이면 newgroup은 {a, e, f} 로 된다.

- MPI_Group_incl

그룹 group의 프로쎄스들중에서 번호가 rank[0], rank[1], … rank[n-1]인 n개 프로쎄스들만으로 이루어지는 새로운 그룹 newgroup을 창조한다. newgroup에서 i번째 프로쎄스는 group에서 rank[i]번째 프로쎄스이다.

문법

```
#include "mpi.h"
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
                                        MPI_Group *newgroup )
```

입구파라메터

group　그룹

n　　　ranks배렬에서의 요소개수(newgroup의 크기)(옹근수)

ranks　newgroup으로 넘어가는 프로쎄스들의 번호배렬(옹근수배렬)

출구파라메터

newgroup : 순서가 ranks에서의 순서대로 되여있는 새로운 그룹

[실례 1.4] group={a, b, c, d, e, f} 이고 rank={3, 1, 2} 이라면 newgroup={d, b, c} 이다.

-MPI_Group_range_excl

현존 그룹으로부터 지정된 범위의 프로세스들을 제외하여 새로운 그룹을 창조한다.

문법

```
#include "mpi.h"
int MPI_Group_range_excl(MPI_Group group, int n, int
                         ranges[][3], MPI_Group *newgroup )
```

입구파라메터

group 그룹(손잡이)

n 배렬 ranks에서의 항목수(옹근수)

ranges 3 옹근수조(첫번째 번호, 마지막 번호, 걸음)의 1 차원 배렬로서 출구그룹 newgroup에서 제외되는 번호들을 지정한다.

출구파라메터

newgroup: 그룹 group에서 순서를 보존하고있는 새로운 그룹

[실례 1.5] group={a, b, c, d, e, f, g ,h, i, j}
ranges={(6, 7, 1) , (1, 6, 2), (0, 9, 4)}의 값들로 이 함수를 호출하면 3 옹근수조목록에 없는 원소값들로 이루어진 새로운 그룹 newgroup={c, j}를 준다.

- MPI_Group_range_incl

현존 그룹에서 지정된 범위의 번호들로 새로운 그룹을 창조한다.

문법

```
#include "mpi.h"
int MPI_Group_range_incl( MPI_Group group, int n, int
                         ranges[][3], MPI_Group *newgroup )
```

입구파라메터

group 그룹(손잡이)

n　　　배렬에서의 3쌍의 개수(옹근수)

ranges 3 옹근수조(첫번째 번호, 마지막 번호, 걸음)의 1차원 배렬로서 출구그룹 newgroup에 포함되는 번호들을 지정한다.

출구파라메터

newgroup : ranges에 정의된 순서를 가진 새로운 그룹

[실례 1.6]　group={a, b, c, d, e, f, g ,h, i, j} , ranges={ (6, 7, 1) , (1, 6, 2), (0, 9, 4)}이라고 하면 첫 3옹근수조 (6, 7, 1)은 번호 (6, 7)인 프로쎄스 {g, h}를 결정하며 둘째 3 쌍 (1, 6, 2)는 번호가 (1, 3, 5)인 프로쎄스{b, d, f}들을 결정하며 세번째 3쌍 (0, 9, 4)는 번호가 (0, 4, 8)인 프로쎄스 {a, e, i}를 결정한다.

결국 이 함수는 새 그룹 newgroup={g, h, b, d, f, a, e, i}를 구축한다.

- MPI_Group_free

그룹을 해방한다.

문법

```
#include "mpi.h"
int MPI_Group_free( MPI_Group *group )
```

입구파라메터

group : 그룹(손잡이)

제 2 절 통신기

1. 통신기의 개념과 기초함수들

MPI프로쎄스들사이의 진행되는 모든 점대점 및 집합통신들은 반드시 통신기라고 하는 기구를 통하여 수행된다. 통신기는 그 자체를 생성하고 리용하며 파괴하는 간단한 규칙들과 속성들을 가진 객체이다. 통신기에 의하여 프로쎄스들사이의 점대점 및 집합통신에 리용되는 통신범위가 결정된다.

MPI에서 진행되는 모든 통신이 반드시 모든 프로쎄스들사이에 이루어지는것은 아니다. MPI프로그람작성자는 자기의 병렬알고리듬의 특성에 따라서 일부 프로쎄스들사이에만 통신하여야 할 필요가 제기될수도 있다. 즉 통신시간을 최대로 줄이고 병렬처리의 효과성을 높이기 위하여 각이한 프로쎄스그룹들로 구성되는 여러가지 통신기를 창조하고 리용할수 있다. 프로쎄스들을 여러 그룹으로 분할하는것은 통보전달의 안정성을 높여준다는 우점도 가진다.

MPI에는 두가지 형태의 통신기(내부통신기, 외부통신기)가 있다.

내부통신기는 개별적인 프로쎄스그룹내의 통신에 리용되는 통신기이다. 이러한 통신기우에서 진행되는 통신을 내부그룹통신이라고 한다.

외부통신기는 사귀지 않는 두 프로쎄스그룹사이의 점대점통신에 리용된다. 이 통신을 외부그룹통신이라고 한다.

통신기와 관련되는 함수들은 다음과 같다.

- MPI_Comm_size

통신기와 련관되는 그룹의 크기를 결정한다.

문법

```
#include "mpi.h"
int MPI_Comm_size( MPI_Comm comm, int *size )
```

입구파라메터:

comm: 통신기(손잡이)

출구파라메터

size : 통신기의 그룹에서 프로쎄스개수(옹근수)

- MPI_Comm_rank

통신기에서 호출하는 프로쎄스의 번호를 결정한다.

문법

```
#include "mpi.h"
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

입구파라메터

comm : 통신기(손잡이)

출구파라메터

rank : 통신기의 그룹에서 호출하는 프로쎄스의 번호(옹근수)

- MPI_Comm_compare

두 통신기를 비교한다.

문법

```
#include "mpi.h"
int MPI_Comm_compare( MPI_Comm  comm1,
                      MPI_Comm  comm2,   int *result)
```

입구파라메터

comm1 통신기 1

comm2 통신기 2

출구파라메터 : result

만일 문맥과 그룹이 같으면 MPI_IDENT, 서로 다른 문맥과 동일한 그룹이면 MPI_CONGRUENT, 서로 다른 문맥이지만 류사한 그룹이면 MPI_SIMILAR, 나머지 경우 MPI_UNEQUAL인 옹근수이다.

2. 통신기구축자

- MPI_Comm_dup

모든 캐쉬정보와 함께 현존 통신기를 복제한다.

문법

```
#include "mpi.h"
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm )
```

입구파라메터

comm : 통신기(손잡이)

출구파라메터

newcomm : 새로운 문맥을 가지지만 comm과 같은 그룹에 있는 새로운 통신기

설명

이 루틴은 새로운 문맥을 가진 새로운 통신기를 창조하며 입구통신기와 꼭같은 프로세스그룹을 포함한다.

- MPI_Comm_create

새로운 통신기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Comm_create( MPI_Comm comm, MPI_Group group,
                                                MPI_Comm   *newcomm )
```

입구파라메터

comm　통신기

group　통신기 comm의 그룹에서 부분그룹(손잡이)

출구파라메터

newcomm : 새로운 통신기(손잡이)

- MPI_Comm_split

색과 열쇠에 기초하여 새로운 통신기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                                        MPI_Comm *newcomm )
```

입구파라메터

comm 통신기(손잡이)

color 부분모임의 할당을 조절(부아닌 옹근수)
key 프로세스번호할당을 조절(옹근수)

출구파라메터
newcomm: 새로운 통신기(손잡이)

설명
color는 MPI_UNDEFINED 또는 부아닌 옹근수이여야 한다.

[실례 1.7] 아래의 값들을 가진 파라메터들로 함수 MPI_Comm_split를 호출한다고 하자.

rank	0	1	2	3	4	5	6	7	8	9
process	a	b	c	d	e	f	g	h	i	j
color	0	x	3	0	3	0	0	5	3	x
key	3	1	2	5	1	1	1	2	1	0

함수는 새로운 통신기들인 그룹 1={f, g, a, d}, 그룹 2={e, i, c}, 그룹 3={h}들을 생성하며 프로세스 b와 j는 그룹이름이 정확치 못하므로 (색갈이 x로서 없음) 여기에 들어가지 못한다.

3. 통신기의 삭제

- MPI_Comm_free
지정된 통신기를 삭제한다.

문법
```
#include "mpi.h"
int MPI_Comm_free( MPI_Comm *comm)
```

입구파라메터
comm : 통신기(손잡이)

이 함수는 이미 창조하였던 통신기가 더는 필요없을 때 사용한다.

4. 내부통신기의 보조함수들

- MPI_Comm_get_name

통신기이름을 돌려준다.

문법

```
#include "mpi.h"
int MPI_Comm_get_name( MPI_Comm comm, char *namep,
                                        int *reslen )
```

입구파라메터

comm : 이름을 얻으려는 통신기(손잡이)

출구파라메터

namep : 적어도 MPI_MAX_NAME_STRING크기의 배렬이여야 한다.
reslen : 통신기이름의 문자개수

- MPI_Comm_group

지정된 통신기와 관련된 그룹에 접근한다.

문법

```
#include "mpi.h"
int MPI_Comm_group( MPI_Comm comm, MPI_Group *group )
```

입구파라메터

comm : 통신기

출구파라메터

group : 통신기의 그룹

- MPI_Comm_remote_group

주어진 외부통신기와 관련되는 원격그룹에 접근한다.

문법

```
#include "mpi.h"
int MPI_Comm_remote_group( MPI_Comm comm, MPI_Group *group )
```

입구파라메터

comm : 통신기(외부통신기이여야 한다.)

출구파라메터

group : 통신기의 원격그룹

- MPI_Comm_remote_size

외부통신기와 관련되는 원격그룹의 크기를 결정한다.

문법

```
#include "mpi.h"
int MPI_Comm_remote_size( MPI_Comm comm, int *size)
```

입구파라메터

comm : 통신기(손잡이)

출구파라메터

size : 통신기의 그룹에서 프로쎄스개수(옹근수)

- MPI_Comm_set_name

통신기에 이름을 설정한다.

문법

```
#include "mpi.h"
int MPI_Comm_set_name( MPI_Comm comm, char *name )
```

입구파라메터

comm : 이름을 설정하여야 할 통신기

name : 통신기의 이름

5. 외부통신기구축자

내부통신기는 MPI_Comm_dup를 호출하여 생성할수 있다. 외부통신기를 구축하기 위해 우선 이전의 그룹들과 꼭같은 그룹으로된 새로운 그룹간통신기를 생성하며 사용자가 정의한 속성들을 넣는다. 외부통신기파괴는 MPI_Comm_free를 호출하여 진행한다. 외부통신기와 관련되는 함수들은 다음과 같다.

- MPI_Intercomm_create

 두개의 내부통신기들로부터 외부통신기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Intercomm_create( MPI_Comm local_comm, int
          local_leader, MPI_Comm peer_comm, int remote_leader,
          int tag, MPI_Comm *newcomm )
```

입구파라메터

local_comm	국부(내부)통신기
local_leader	국부그룹 local_comm의 지휘프로쎄스번호(보통 0)
peer_comm	원격통신기
remote_leader	peer_comm에서 원격지휘프로쎄스번호(보통 0)
tag	통보표적

출구파라메터

 newcomm : 창조된 외부통신기

- MPI_Intercomm_merge

 외부통신기로부터 내부통신기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Intercomm_merge( MPI_Comm comm, int high,
                         MPI_Comm *newcomm )
```

입구파라메터

comm 외부통신기

high 새로운 통신기를 창조할 때 comm내에서 두개의 내부통신기들의 그룹들을 순서화하는데 리용하는 값

출구파라메터

newcomm : 창조된 내부통신기

- MPI_Comm_test_inter

comm이 외부통신기인가를 검사한다.

문법

```
#include "mpi.h"
int MPI_Comm_test_inter( MPI_Comm comm, int *flag )
```

입구파라메터

comm : 통신기(손잡이)

출구파라메터

flag : 기발(론리값)

[실례 1. 8] 실례에서 프로세스들은 3 개의 그룹으로 나누어져있다. 그룹 0 과 1 이 련결되여있고 그룹 1 과 2 가 련결되여있다. 결국 그룹 0 과 2 에는 한개씩의 내부통신기가 필요하지만 그룹 1 에는 2 개의 외부통신기가 필요하다.

아래에서는 매 프로세스의 국부번호와 대역번호(괄호안에)를 주고있다.

그룹 0	그룹 1	그룹 2
0(0)	0(1)	0(2)
1(3)	1(4)	1(5)
2(6)	2(7)	2(8)

3(9)   3(10)   3(11)

이것을 만드는 프로그램을 작성하기로 한다.

```
#include <stdio.h>
#include  "mpi.h"

main(int argv, char **argc)
  {
    MPI_Comm myComm;          /* 국부부분그룹의 내부통신기 */
    MPI_Comm myFirstComm;       /* 외부통신기 */
    MPI_Comm mySecondComm;   /* 두번째 외부통신기 */
    int membershipKey, rank;

    MPI_Init(&argc, &argv);   /* MPI 서고초기화 */
      /* 자기의 번호를 결정 */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      /* membershipKey에서 번호 [0, 1, 2] 발생 */
    MembershipKey=rank%3;
     /* 국부부분그룹에 대한 내부통신기의 구축 */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank,
                         &myComm);
     /* 외부통신기들의 구축 */
    if (membershipKey == 0) {
/* 그룹0이 그룹1과 련결 */
        MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
            01, &myFirstComm);
    }
else if (membershipKey == 1) {
/* 그룹0과 그룹1,2가 련결 */
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
            01, &myFirstComm);
      MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
            12, &mySecondComm);
```

```
    }
   else if (membershipKey == 2) {
 /* 그룹2와 그룹1이 련결 */
       MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
            12, &myFirstComm);
    }

 /**************     계산부분 *******************/

 /* 3개의 통신기삭제 */
 MPI_Comm_free(&myComm);
 MPI_Comm_free(&myFirstComm);
 if (membershipKey == 1)
    MPI_Comm_free(&mySecondComm);
 MPI_Finalize();
return 0;
}
```

제 3 절 가상위상

1. 가상위상의 개념

프로쎄스들사이의 통신모형을 그라프로 표시할수 있다. 그라프에서 마디는 프로쎄스를, 변은 두개 프로쎄스사이의 호상통신을 표시한다. 모든 통신들은 거의나 대칭이므로 통신그라프는 대칭그라프이다.

MPI는 그룹에 있는 임의의 프로쎄스쌍사이의 통보전송을 보장한다.

프로그람에서 통보전송을 정점이 프로쎄스이고 그라프변이 통보인 방향그라프로 표시할수 있다.

가상위상이란 고리, 살창, 별, 나무 등 임의로 주어지는 그라프형태를 프로그람적으로 실현한 위상이다. 가상위상은 통신기와 련관된 프로쎄스들의 이름달기기구를 매우 편리하게 보장하며 프로쎄스들을 체계의 장치에로 넘기는 강력한 수단이다. MPI에서 가상위상은 통신기에 련결된 프로쎄스그룹에서만 제시할수 있다.

이미 지적한것처럼 프로세스그룹은 n개 프로세스로 이루어진 조로서 매 프로세스에는 0 ~ (n-1)사이의 옹근수번호가 대응된다.

대다수의 병렬계산환경들에서 선형적인 프로세스번호달기는 프로세스들사이의 론리적인 통신모형을 민감하게 반영하지 못한다.

병렬알고리듬들은 보통 2 차원 혹은 체적살창형태의 위상모형으로 표현되며 일반적으로 프로세스들의 론리적인 배치는 임의의 그라프로 표시되여야 한다.

프로세스들의 가상위상과 물리적장치의 위상을 구분하여야 한다.

가상위상기구는 병렬프로그람의 서술을 현저하게 단순화하고 쉽게 하며 프로그람을 리해하기 쉽게 한다.

사용자는 처리기들의 물리적련관도식이 아니라 프로세스들의 가상위상만을 서술하면 될것이다.

2. 데카르트위상함수들

2.1 데카르트위상구축함수

- MPI_Cart_create

데카르트위상정보가 첨부되는 새로운 통신기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Cart_create( MPI_Comm comm_old, int ndims, int
        *dims, int *periods, int reorder, MPI_Comm *comm_cart )
```

입구파라메터

comm_old - 입구통신기

ndims - 데카르트살창의 차원수

dims - 매 차원에 있는 모든 프로세스개수를 지정하는 ndims크기의 옹근수배렬

periods - 매 차원에서 살창이 주기적인가(참) 아닌가(거짓)를 지정하는 ndims크기의 론리값배렬

reorder - 프로세스번호달기가 재순서화되는가(참) 아닌가(거짓)를 지정(론리값)

출구파라메터

　comm_cart　- 데카르트위상을 가지는 새로운 통신기

2.2　데카르트살창제시함수

- MPI_Dims_create

데카르트살창에서 처리기들의 분할을 진행한다.

문법

```
#include "mpi.h"
int MPI_Dims_create( int nnodes,  int ndims,  int *dims)
```

입구파라메터

　nnodes 살창에 있는 마디점수(옹근수)

　ndims　데카르트차원수(옹근수)

　dims : 매 차원에서 마디점수를 지정하는 ndims 크기의 옹근수배렬

출구파라메터

　dims : 매 차원에서 마디점수를 지정하는 ndims 크기의 옹근수배렬

　dims는 입구인 동시에 출구로 된다.

[실례 2.1] MPI_Dims_create함수를 호출한 실례들

호출전dims	호출	호출후의 dims
(0, 0)	MPI_Dims_create(6, 2, dims)	(3, 2)
(0, 0)	MPI_Dims_create(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_Dims_create(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_Dims_create(7, 3, dims)	입구값오유

2.3　데카르트정보함수들

- MPI_Cartdim_get

통신기와 련관되는 데카르트위상의 차원수를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Cartdim_get( MPI_Comm comm,   int *ndims )
```

입구파라메터

comm: 데카르트구조를 가진 통신기

출구파라메터

ndims : 데카르트구조의 차원수(옹근수)

- MPI_Cart_get

통신기와 관련한 데카르트위상정보를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Cart_get(MPI_Comm comm, int maxdims,   int *dims,
                                        int *periods, int *coords )
```

입구파라메터

comm - 데카르트구조를 가진 통신기

maxdims - 호출하는 프로그람에서 dims, periods, coords배렬들의 최대길이

출구파라메터

dims - 매개 데카르트차원에 있는 프로쎄스개수(옹근수배렬)

periods - 매 자리표에서 주기성(true/false)을 지정하는 값배렬

coords - 데카르트구조에서 호출하는 프로쎄스의 자리표

2.4 데카르트변환함수

- MPI_Cart_rank

지정된 데카르트자리표에 있는 프로쎄스의 번호를 결정한다.

문법

```
#include "mpi.h"
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank )
```

입구파라메터

comm 데카르트구조의 통신기

coords 옹근수배렬(크기는 ndims)로서 프로쎄스의 데카르트자리표를 지정한다.

출구파라메터

rank : 지정된 프로쎄스의 번호(옹근수)

- MPI_Cart_coords

그룹안에서 지정된 처리기의 데카르트위상자리표를 결정한다.

문법

```
#include "mpi.h"
int MPI_Cart_coords( MPI_Comm comm, int rank, int maxdims,
                                        int  *coords )
```

입구파라메터

comm - 데카르트구조를 가진 통신기

rank - comm의 그룹내부에서 프로쎄스의 번호

maxdims -호출하는 프로그람에서 dims, periods, coords의 최대길이

출구파라메터

coords - 지정된 프로쎄스의 데카르트자리표를 포함하는 ndims크기의 옹근수배렬

2.5 데카르트밀기, 분할, 넘기기함수

- MPI_Cart_shift

밀기방향과 거리가 주어지면 밀기된 원천과 목적번호들을 돌려준다.

문법

```
#include "mpi.h"
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ,
                    int *source, int *dest )
```

입구파라메터

comm 데카르트구조의 통신기

direction 밀기할 자리표차원(옹근수)

displ 변위방향 (> 0: 우로 밀기, < 0: 아래로 밀기) (옹근수)

출구파라메터

source 원천프로쎄스의 번호(옹근수)

dest 목적프로쎄스의 번호(옹근수)

- MPI_Cart_sub

더 낮은 차원의 데카르트부분살창을 이루는 부분그룹들로 통신기를 분할한다.

문법

```
#include "mpi.h"
int MPI_Cart_sub( MPI_Comm comm, int *remain_dims,
                  MPI_Comm *comm_new )
```

입구파라메터

comm 데카르트구조의 통신기

remain_dims remain_dims의 i번째 요소는 i번째 차원이 부분살창에 들어가는가(참), 들어가지 않는가(거짓)를 결정한다.

출구파라메터

comm_new : 호출프로쎄스를 포함하는 부분살창에 해당한 통신기

[실례 2.2] MPI_Cart_create(…, comm)이 (2 * 3 * 4)살창을 창조하였다고 하자. remain_dims=(true, false, true)값으로 MPI_Cart_sub(comm, remain_dims, comm_new)를 호출하면 3개의 통신기가 생성되며 매개

통신기에 데카르트위상이 2＊4 인 8개의 프로세스들이 들어간다.

만일 remain_dims=(false, false, true)로 함수를 호출하면 한개 차원의 데카르트살창에 4 개의 프로세스들을 가지는 사귀지 않는 6 개의 통신기가 생성된다.

- MPI_Cart_map

프로세스를 데카르트위상에로 넘긴다.

문법

```
#include "mpi.h"
int MPI_Cart_map( MPI_Comm comm,   int ndims,   int *dims,
                  int *periods,      int *newrank)
```

입구파라메터

comm 입구통신기

ndims 데카르트구조의 차원수(옹근수)

dims 매 자리표방향에서의 프로세스개수를 지정하는 ndims크기의 옹근수배렬

periods 매 자리표방향에서의 주기성을 지정하는 ndims크기의 론리배렬

출구파라메터

newrank : 재순서화된 호출프로세스의 번호, 호출프로세스가 살창에 속하지 않으면 MPI_UNDEFINED의 옹근수값

3. 그라프위상함수들

여기서는 그라프위상구조와 관련되는 MPI함수들을 서술한다.

3.1 그라프구축함수와 그라프요구함수들

- MPI_Graph_create

그라프위상정보가 첨부되는 새로운 통신기를 만든다.

문법

```
#include "mpi.h"
int MPI_Graph_create( MPI_Comm comm_old, int nnodes, int
     *index, int *edges, int reorder, MPI_Comm *comm_graph )
```

입구파라메터

comm_old	위상이 없는 입구통신기
nnodes	그라프에서의 마디수(옹근수)
index	마디점의 준위들을 서술하는 옹근수배렬
edges	그라프변들을 서술하는 옹근수배렬
reorder	번호재순서화를 하는가(참), 안하는가(거짓)를 지정(론리값)

출구파라메터

comm_graph : 그라프위상이 첨부된 새로운 통신기

[실례 2.3] 린접행렬이 다음과 같은 4 개의 프로세스 0, 1, 2, 3 이 있다고 하자.

프로세스	린접
0	1, 3
1	0
2	3
3	0, 2

이때 입구파라메터들은 다음과 같이 결정한다.

nnodes=4
index=(2, 3, 4, 6)
edges=(1, 3, 0, 3, 0, 2)

- MPI_Graphdims_get

통신기와 련관된 그라프위상정보를 준다.

문법

```
#include "mpi.h"
int MPI_Graphdims_get( MPI_Comm comm, int *nnodes, int
```

*nedges)

입구파라메터

comm : 그라프구조의 그룹을 가진 통신기

출구파라메터

nnodes 그라프의 마디점수(옹근수)

nedges 그라프의 변개수(옹근수)

- MPI_Graph_get

통신기와 련관된 그라프위상정보를 준다.

문법

```
#include "mpi.h"
int MPI_Graph_get( MPI_Comm comm, int maxindex, int
                  maxedges, int *index, int *edges )
```

입구파라메터

comm 그라프구조를 가진 통신기

maxindex 호출프로그람에서의 벡토르 index의 길이(옹근수)

maxedges 호출프로그람에서의 벡토르 edges의 길이(옹근수)

출구파라메터

index 그라프구조를 담은 옹근수배렬(MPI_Graph_create참고할것)

edges 그라프구조를 담은 옹근수배렬

3.2 그라프넘기기함수와 위상요구함수

- MPI_Graph_map

프로쎄스를 그라프위상정보에로 넘긴다.

문법

```
#include "mpi.h"
int MPI_Graph_map(MPI_Comm comm, int nnodes, int
```

*index, int *edges, int *newrank)

입구파라메터

comm	입구통신기
nnodes	그라프마디개수(옹근수)
index	그라프구조를 지정하는 옹근수배렬(MPI_Graph_create 함수를 참고할것)
edges	그라프구조를 지정하는 옹근수배렬

출구파라메터

newrank : 재순서화된 호출프로세스의 번호, 호출프로세스가 그라프에 속하지 않으면 MPI_UNDEFINED인 옹근수값

- MPI_Topo_test

통신기와 련관된 위상형을 검사한다.

문법

```
#include "mpi.h"
int MPI_Topo_test( MPI_Comm comm, int *top_type )
```

입구파라메터

comm : 통신기

출구파라메터

top_type : 통신기 comm의 위상형

다음의 3개값들중의 하나를 준다.

MPI_GRAPH : 그라프위상

MPI_CART : 데카르트위상

MPI_UNDEFINED : 위상이 없음

3.3 그라프정보함수들

- MPI_Graph_neighbors_count

그라프위상을 따르는 마디의 린접들의 개수를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Graph_neighbors_count( MPI_Comm comm, int rank,
                int *nneighbors )
```

입구파라메터

comm 그라프위상을 가진 통신기

rank 그룹 comm에서 프로쎄스의 번호(옹근수)

출구파라메터

nneighbors : 지정된 프로쎄스의 린접개수(옹근수)

- MPI_Graph_neighbors

그라프위상을 따르는 마디의 린접들을 돌려준다.

문법

```
#include "mpi.h"
int MPI_Graph_neighbors( MPI_Comm comm, int rank, int
                         maxneighbors, int *neighbors )
```

입구파라메터

comm 그라프위상을 가진 통신기(손잡이)

rank 그룹 comm에서의 프로쎄스의 번호(옹근수)

maxneighbors 배렬 neighbors의 길이(옹근수)

출구파라메터

neighbors : 주어진 프로쎄스의 린접프로쎄스들의 번호(옹근수배렬)

제 4 절 점대점(point-to-point)연산

MPI에서 프로쎄스들사이의 기본통신기구는 호상작용하는 프로쎄스쌍들

사이의 자료전송으로서 하나는 송신프로쎄스이고 다른 하나는 수신프로쎄스이다. 이 통신을 점대점(point-to-point)연산이라고 한다.

MPI의 거의 모든 연산들은 이 점대점연산에 기초하여 구축되였다.

MPI는 표적(tag)이 붙은 자료형(datatype)을 전송하도록 하는 여러가지 송수신함수들을 제공한다.

MPI에는 점대점연산을 수행하는 여러가지 함수들이 있다.

기초적인 점대점송신 및 수신연산에는 차단연산과 비차단연산이 있다. 또한 송신연산에는 완충형, 동기형, 준비상태형이 있다. 수신연산에는 차단과 비차단 두가지이며 이 매개 수신연산은 모든 송신연산과 서로 대응된다.

1. 차단통신

1.1 차단자료전송

차단전송함수들은 비동기형으로서 파라메터전송후에 체계에 조종을 넘기며 수신측에서 통보를 받았는가 받지못하였는가는 송신측에서 고려하지 않는다. 송신측에서 전송함수호출후에 전송할 자료렬에 대한 접근(읽기 혹은 쓰기)이 있을 때 통보자료들이 체계의 중간완충기에 안전하게 보관될 때까지 즉 송신완충기에 다시 접근(읽기 혹은 쓰기)가능할 때까지는 그 접근이 차단된다.

수신측에서는 이 경우에 어떤 예견도 없으며 송신측에서는 대응하는 수신함수가 시동되였는가에 무관계하게 송신을 끝낸다.

응용프로그람에서는 전송자료를 체계완충기에 보관하도록하는 그 어떤 서술도 필요없으며 전송자료들은 체계에 의하여 체계완충기에 완충된다.

차단전송함수들로는 표준전송함수(MPI_Send), 완충전송함수(MPI_Bsend), 동기전송함수(MPI_Ssend) 및 준비전송함수(MPI_Rsend)들이 있다.

- MPI_Send

기본송신을 진행한다.

문법

```
#include "mpi.h"
int MPI_Send( void *buf, int count, MPI_Datatype datatype,
                        int dest, int tag, MPI_Comm comm )
```

입구파라메터

buf　　　송신완충기의 시작주소
count　　송신완충기의 요소수(부아닌 옹근수)
datatype 매 송신완충기요소의 자료형(손잡이)
dest　　　목적프로쎄스번호(옹근수)
tag　　　통보표적(옹근수)
comm　　통신기

- MPI_Bsend

사용자가 정의한 완충에 의한 기본전송함수이다.

문법

```
#include "mpi.h"
int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int
                dest, int tag, MPI_Comm comm )
```

입구파라메터

buf　　　- 송신완충기의 시작주소
count　　- 송신완충기에서 요소의 수
datatype - 매 송신완충기요소의 자료형
dest　　　- 목적프로쎄스번호
tag　　　- 통보표적
comm　　- 통신기

설명

이 전송은 사용자가 송신통보가 어디에 완충될것인가를 고려하지 않고도 통보를 전송하는데 편리하다. 사용자가 MPI_Buffer_attach함수를 리용하여 완충기공간을 제공한다.

C에서는

MPI_Buffer_detach(&b, &n);

포트란에서는

MPI_Buffer_attach(b, n);

에 의하여 통보가 배정될수 있다.

- MPI_Ssend

기본동기전송함수이다.

문법

```
#include "mpi.h"
int MPI_Ssend( void *buf, int count, MPI_Datatype datatype,
                    int dest, int tag, MPI_Comm comm )
```

입구파라메터

buf	송신완충기의 시작주소
count	송신완충기의 요소수(부아닌 옹근수)
datatype	매 송신완충기요소의 자료형(손잡이)
dest	목적프로쎄스번호
tag	통보표적
comm	통신기

- MPI_Rsend

기본기다림전송함수이다.

문법

```
#include "mpi.h"
int MPI_Rsend( void *buf, int count, MPI_Datatype datatype,
                                    int dest, int tag, MPI_Comm comm )
```

입구파라메터

buf	송신완충기의 초기주소
count	송신완충기의 요소수(부아닌 옹근수)
datatype	매 송신완충기요소의 자료형(손잡이)
dest	목적프로쎄스번호
tag	통보표적
comm	통신기

1.2 차단수신

자료차단수신함수(MPI_Recv)는 수신자료완충기가 대응하는 통보를 얻은 후에만 조종을 되돌린다.

- MPI_Recv

기본수신함수이다.

문법

```
#include "mpi.h"
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status )
```

출구파라메터

buf   수신완충기의 시작주소

status 수신상태정보

입구파라메터

count   수신해야 할 요소개수(옹근수)

datatype 매 수신완충기요소의 자료형(손잡이)

source   송신(원천)프로쎄스번호(옹근수)

tag   통보표적(옹근수)

comm   통신기

프로쎄스들사이의 송수신을 진행함에 있어서 지켜야 할 문제들을 실례를 통하여 보기로 한다.

아래의 두 실례들은 송신측과 수신측의 자료형들이 일치하여야 한다는것을 보여주고있다.

[실례 4.1] 송신측 및 수신측에 대한 프로그람부분을 보기로 하자.

```
            …
    MPI_Comm_rank(comm, &rank);
    if (rank == 0)
```

```
        MPI_Send(a, 10, MPI_FLOAT, 1, tag, comm);
    else if (rank == 1)
        MPI_Recv(b, 10, MPI_FLOAT, 0, tag, comm, status);
                    …
```

이 코드에서 a와 b가 10 차이상의 류점수배렬이면 오유가 없는것으로 된다. 만일 송수신함수들에서 송수신자료형을 나타내는 3 번째 파라메터가 류점수가 아닌 다른 형, 실례로 옹근수형이면 a, b는 10 차이상의 옹근수배렬이여야 한다.

[실례 4.2] 송신측과 수신측이 형대응을 틀리게 한 실례

```
                    …
    MPI_Comm_rank(comm, &rank);
    if (rank == 0)
        MPI_Send(a, 40, MPI_FLOAT, 1, tag, comm);
    elseif (rank == 1)
        MPI_Recv(b, 40, MPI_BYTE, 0, tag, comm, status);
                    …
```

우에서 송신자료형은 MPI_FLOAT이고 수신자료형은 MPI_BYTE이므로 오유이다.

다음 3 가지 실례프로그람들은 한 처리기에서 송신함수와 수신함수를 동시에 호출하는 경우를 보여준다.

[실례4.3] 통보교환

```
  …
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
   MPI_Send(sendbuf, count, MPI_FLOAT, 1, tag, comm);
   MPI_Recv(recvbuf, count, MPI_FLOAT, 1, tag, comm, &status);
 }
 if (rank == 1) {
   MPI_Recv(recvbuf, count, MPI_FLOAT, 0, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_FLOAT, 0, tag, comm);
```

```
 }
    …
```

이 프로그람은 완충기를 위한 기억구역이 확보되지 않은 경우에도 정확한 통신을 진행한다.

[실례 4.4] 통보교환

```
    …
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
   MPI_Recv(recvbuf, count, MPI_FLOAT, 1, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_FLOAT, 1, tag, comm);
}
if (rank == 1) {
   MPI_Recv(recvbuf, count, MPI_FLOAT, 0, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_FLOAT, 0, tag, comm);
}
    …
```

0 번 프로쎄스의 수신함수는 1 번 프로쎄스로부터 우선 자료를 받고 다음에 1 번 프로쎄스에 자료를 보내야 하며 오직 1 번 프로쎄스가 0 번 프로쎄스에로의 자료전송을 완료한 후에야 완료할수 있다. 1 번 프로쎄스에 대해서도 마찬가지이며 결국 프로그람은 교착상태에 빠지게 된다.

[실례 4.5] 완충화를 예견한 통보교환

```
    …
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
   MPI_Send(sendbuf, count, MPI_FLOAT, 1, tag, comm);
   MPI_Recv(recvbuf, count, MPI_FLOAT, 1, tag, comm,  &status);
 }
 if (rank == 1) {
   MPI_Send(sendbuf, count, MPI_FLOAT, 0, tag, comm);
   MPI_Recv(recvbuf, count, MPI_FLOAT, 0, tag, comm, &status);
 }
    …
```

매 프로쎄스에서 송신함수들이 완료하고 수신함수들이 시동하도록 송신통보를 어디엔가 복사하여야 한다. 프로그람을 안전하게 끝내기 위하여서는 적어도 두 통보중의 하나는 완충화되여있어야 한다.

통신체계가 자기의 내부완충기에 통보를 완충만 한다면 제대로 돌아갈것이며 그렇지 않은 경우 교착상태에 빠진다. 결국 실행시에 완충기기억량에 의해 프로그람의 완료가 좌우된다. 자료량이 많지 않으면 프로그람은 정확히 실행된다.

2. 비차단통신

모든 비차단통신조작들은 통신요구를 체계에 제출하는 함수와 그 통신요구가 끝났는가를 응용프로그람이 검사하도록 하는 "완료검사" 함수 두개로 항상 나누어져있다. 함수에서 앞붙이 B, S, R는 각각 완충형, 동기형, 준비상태형전송함수라는것, 그리고 앞붙이 I는 연산이 비차단형이라는것을 가리킨다. 비차단수신함수는 MPI_Irecv함수 하나뿐이며 이 모든 비차단연산들은 완료기다림 및 검사함수에 의하여 끝난다.

2.1 비차단자료전송

비차단함수는 체계가 송신완충기에 대해서 자료를 복사할수 있게 되였다는것을 지적한다. 파라메터들을 체계에 전송한 후에 함수는 조종을 되돌린다. 그 다음에 전송프로쎄스는 송신완충기에 접근(읽기 또는 쓰기)하지 말아야 한다. 접근하는 경우 체계는 전송자료의 타당성을 담보하지 않는다. 고찰하는 연산의 완료를 기다리고 검사하기 위하여 MPI_Wait함수와 MPI_Test함수를 리용한다. 여기서 완료란 전송통보들이 체계의 중간완충기에 안전하게 들어가고 송신완충기가 접근가능하게 되였을 때이다.

비차단전송함수들에는 표준전송함수(MPI_Isend), 완충전송함수(MPI_Ibsend), 동기전송함수(MPI_Issend) , 준비상태따름전송함수(MPI_Irsend) 등이 있다.

- MPI_Isend

비차단전송을 진행한다.

문법

#include "mpi.h"
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)

입구파라메터

buf 송신완충기의 시작주소
count 송신완충기의 요소수(옹근수)
datatype 매 송신완충기요소의 자료형
dest 목적프로쎄스의 번호(옹근수)
tag 통보표적(옹근수)
comm 통신기(손잡이)

출구파라메터

request: 통신요구(손잡이)

- MPI_Ibsend

비차단완충전송을 진행한다.

문법

#include "mpi.h"
int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

입구파라메터

buf 송신완충기의 시작주소
count 송신완충기의 요소수(옹근수)
datatype 매 송신완충기요소의 자료형
dest 목적프로쎄스의 번호(옹근수)
tag 통보표적(옹근수)
comm 통신기(손잡이)

출구파라메터

request : 통신요구(손잡이)

- MPI_Issend

비차단동기전송을 진행한다.

문법

```
#include "mpi.h"
int MPI_Issend( void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request )
```

입구파라메터

buf	송신완충기의 시작주소
count	송신완충기의 요소수(옹근수)
datatype	매 송신완충기요소의 자료형
dest	목적프로쎄스의 번호(옹근수)
tag	통보표적(옹근수)
comm	통신기(손잡이)

출구파라메터

request : 통신요구(손잡이)

- MPI_Irsend

비차단준비전송을 진행한다.

문법

```
#include "mpi.h"
int MPI_Irsend( void *buf, int count, MPI_Datatype datatype,
            int dest, int tag, MPI_Comm comm, MPI_Request *request )
```

입구파라메터

buf	송신완충기의 시작주소
count	송신완충기의 요소수(옹근수)
datatype	매 송신완충기요소의 자료형
Dest	목적프로쎄스의 번호(옹근수)

tag　　　통보표적(옹근수)
comm　　통신기(손잡이)

출구파라메터
request : 통신요구(손잡이)

2.2 비차단자료수신

비차단수신함수 MPI_Irecv는 체계가 수신완충기에 자료를 쓰기시작할수 있다는것을 지적하며 함수파라메터들을 체계에 넘긴 후에 조종을 되돌린다.

- MPI_Irecv

비차단수신을 진행한다.

문법

```
#include "mpi.h"
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm comm, MPI_Request *request )
```

입구파라메터

buf　　　수신완충기의 시작주소
datatype 매 수신완충기요소의 자료형(손잡이)
source　원천프로쎄스번호(옹근수)
tag　　　통보표적(옹근수)
comm　　통신기
count　　송신완충기의 요소수(옹근수)

출구파라메터
request : 통신요구(손잡이)

[실례 4.6] 동기 및 준비상태 자료교환수법들의 리용

```
float buff [1000][ 2];
MPI_Status status;
MPI_Comm comm;
MPI_Request req;
        …
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
    MPI_Irecv(&buff[1][1], 1000, MPI_FLOAT, 1, 1, comm, &req);
    MPI_Irecv(&buff[1][2], 1000, MPI_FLOAT, 1, 2, comm, &req);
    MPI_Waitall(2, &req, &status);
}
else if (rank == 1) {
   MPI_Ssend(&buff[1][2], 1000, MPI_FLOAT, 0, 2, comm, &status);
   MPI_Ssend(&buff[1][1], 1000, MPI_FLOAT, 0, 1, comm, &status);
       }
    …
```

1 번 프로쎄스의 첫 동기전송함수는 0 번 프로쎄스의 두번째 수신함수에 대응한다. 이 송신함수는 0 번 프로쎄스의 두번째 수신함수가 시동한 후에만 그리고 0 번 프로쎄스의 첫번째 수신함수의 초기화가 끝난 후에야 끝날것이다.

2.3 비차단통신조작들의 완료함수들

MPI_Wait함수와 MPI_Test함수들은 비차단송수신을 완료하는데 리용한다.

전송처리의 완료란 전송프로쎄스에서 송신완충기에 대한 접근이 가능하게 되였을 때이며 수신처리의 완료란 수신완충기에 통보가 들어왔고 수신완충기는 접근가능하다는것을 의미한다.

- MPI_Wait

MPI비차단송신과 수신처리가 끝나기를 기다린다.

문법

```
#include "mpi.h"
```

```
int MPI_Wait( MPI_Request *request, MPI_Status *status)
```

입구파라메터
  request : 요구(손잡이)

출구파라메터
  status : 상태객체

- MPI_Test
  송신 또는 수신이 끝났는가를 검사한다.

문법
```
#include "mpi.h"
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status)
```

입구파라메터
 request : 통신요구(손잡이)

출구파라메터
  flag   연산이 성공하면 참(론리값)
  status 상태객체

[실례 4.7] 생산자는 많고 소비자는 하나인 경우에 비차단통신을 리용한다.

```
typedef struct {
    char data[MAXSIZE];
    int datasize;
    MPI_Request req;
} Buffer;
int size, rank;
Buffer buffer[];
MPI_Status status;
   …
/* 매 프로세스는 체계의 콤퓨터수와 자기 번호를 결정한다. */
```

```
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
/* 생산자 */
if (rank != size-1) {
   /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(buffer));
   /* 주 순환 */
    while (1) {
        /* 자료파일의 생성과 완충기에 보존된 바이트수의 복귀 */
         produce(buffer->data, buffer->datasize);
        /* 자료전송 */
         MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
                    size-1, tag, comm);
    }
}
 /* rank=size-1 인 경우(소비자) */
else {
    /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(Buffer)*(size-1));
    for (i=0; i <= size-1;   i++)
       /* 매 생산자로부터의 수신   */
     MPI_Irecv(buffer[i].data,  MAXSIZE,  MPI_CHAR,  i,  tag,  comm,
&(buffer[i].req));
  /* 주 순환 */
   for (i=0; i <= size-1; i++) {
     MPI_Wait(&(buffer[i].req), &status);
     /* 실지 수신된 바이트수를 결정 */
     MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
     /* 자료요소수신완충기 */
     consume(buffer[i].data, buffer[i].datasize);
    /* 새로운 수신 */
    MPI_Irecv(buffer[i].data,   MAXSIZE,   MPI_CHAR, i, tag,
             comm, &(buffer[i].req));
  }
```

```
}
```

매 생산자는 무한순환으로 돌아가며 매 순환에서는 하나의 통보를 생산하고 그것을 전송한다. 소비자는 매 생산자와 접속하여 통보를 받는다.

[실례 4.8] 생산자는 많고 소비자는 하나인 경우 MPI_Test함수를 리용한 비차단통신실례

```
typedef struct {
      char data[MAXSIZE];
      int datasize;
      MPI_Request req;
}Buffer;

Buffer buffer[];
MPI_Status status;
    …
/* 매 프로쎄스는 체계의 콤퓨터수와 자기 번호를 결정한다. */
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
/* 생산자 */
if (rank != size-1) {
      /* 완충기의 생성 */
      buffer=(Buffer *)malloc(sizeof(buffer));
      /* 주 순환 */
      while(1) {
            /* 자료파일의 생성과 완충기에 보존된 바이트수의 복귀 */
             produce(buffer->data, buffer->datasize);
            /* 자료전송 */
             MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
                             size-1, tag, comm);
       }
 }
/* rank=size-1 ; 소비자 */
else {
```

```
    /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(Buffer)*(size-1));
    for (i=0; i < size-1;  i++)
       /* 매 생산자로부터의 수신 */
      MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
&(buffer[i].req));
  /* 주 순환 */
 i=0;
 while (1) {
   for (flag=0; !flag; i=(i+1)%(size-1)) {
     /* 수신완료검사 */
     MPI_Test(&(buffer[i].req), &flag, &status);
    /* 실지 수신된 바이트수를 결정 */
    MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
    /* 소비자함수호출 */
    consume(buffer[i].data, buffer[i].datasize);
    /* 새로운 수신  */
    MPI_Irecv(buffer[i].data,  MAXSIZE,  MPI_CHAR, i,  tag,
             comm, &(buffer[i].req));
   }
 }
}
```

여기에서 만일 생산자로부터 오는 통보가 없으면 소비자프로쎄스는 이전 생산자에게로 돌아간다. 우의 프로그람의 MPI_Wait대신에 MPI_Test를 리용하면 "선래선봉사"형식으로 동작한다.

아래에서는 비차단연산들의 의미에 대하여 구체적으로 본다.

[실례 4.9]

```
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
   MPI_Isend(a, 1, MPI_FLOAT, 1, 0, comm, &r1);
   MPI_Isend(b, 1, MPI_FLOAT, 1, 0, comm, &r2);
}
```

```
else if (rank == 1) {
    MPI_Irecv(a, 1, MPI_FLOAT, 0, 0, comm, &r1);
    MPI_Irecv(b, 1, MPI_FLOAT, 0, 0, comm, &r2);
}
MPI_Wait(&r2, &status);
MPI_Wait(&r1, &status);
```

0 번 프로세스의 첫번째 송신은 1 번 프로세스의 첫 수신과 정합된다. 순서결정은 송신연산과 수신연산이 정합될것을 요구한다.

[실례 4.10] 비차단연산들의 완료순서

```
MPI_Comm_rank(comm, &rank);
flag1=false;
flag2=false;
if (rank == 0) {
    MPI_Isend(a, n, MPI_FLOAT, 1, 0, comm, &r1);
    MPI_Isend(b, 1, MPI_FLOAT, 1, 0, comm, &r2);
    while(!(flag1 && flag2)) {
        if (! flag1)
          MPI_Test(&r1, &flag1, &s);
        if (! flag2)
          MPI_Test(&r2, &flag2, &s);
    }
}
else {
  if (rank == 1) {
    MPI_Irecv(a, 1, MPI_FLOAT, 0, 0, comm, &r1);
    MPI_Irecv(b, 1, MPI_FLOAT, 0, 0, comm, &r2);
     while(!(flag1 && flag2)) {
        if (! flag1)
          MPI_Test(&r1, &flag1, &s);
        if (! flag2)
          MPI_Test(&r2, &flag2, &s);
```

```
        }
    }
}
```

실례 4.9 에서와 같이 0 번 프로쎄스의 첫째 전송은 1 번 프로쎄스의 첫 수신과 정합된다. 그러나 두번째 수신은 첫번째 수신전에 끝날수 있고 두번째 전송은 첫번째 전송전에 끝날수 있다. 특히 첫번째 자료교환의 량이 두번째 자료교환의 량보다 더 많을 때 이러한 현상이 일어난다.

2.4 비차단연산들의 다중완료

한번의 호출로 여러번 호출된 비차단통신조작들을 동시에 끝내는것은 하나씩 끝내는것보다 훨씬 편리하고 효과적이다.

MPI_Waitany함수 혹은 MPI_Testany함수는 임의의 한개 비차단연산을 끝내는데, MPI_Waitall함수 혹은 MPI_Testall함수는 이미 시작된 모든 비차단연산들을 끝내는데, MPI_Waitsome함수와 MPI_Testsome함수는 일부 비차단연산들을 끝내는데 리용한다.

- MPI_Waitall

주어진 모든 통신이 끝나기를 기다린다.

문법

```
#include "mpi.h"
int MPI_Waitall(int count,  MPI_Request array_of_requests[],
                            MPI_Status array_of_statuses[] )
```

입구파라메터

count	목록길이(옹근수)
array_of_requests	통신요구배렬(손잡이배렬)

출구파라메터

array_of_statuses : 통신상태객체배렬(상태배렬)

- MPI_Waitany

임의의 한개 송신 혹은 수신이 끝나기를 기다린다.

문법

```
#include "mpi.h"
int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int *index, MPI_Status *status )
```

입구파라메터

count 목록길이(옹근수)

array_of_requests 요구배렬(손잡이배렬)

출구파라메터

index 완성된 연산에 대한 손잡이첨수(옹근수), 범위는 0 부터 count-1 사이의 값이다.

status 객체상태

설명

만일 모든 요구가 MPI_REQUEST_NULL이면 index는 MPI_UNDEFINED 값으로서 복귀되며 상태 status는 빈 상태로 복귀된다.

- MPI_Waitsome

일부 주어진 통신들이 끝나기를 기다린다.

문법

```
#include "mpi.h"
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
        int *outcount, int array_of_indices[], MPI_Status
                                    array_of_statuses[] )
```

입구파라메터

incount array_of_requests의 길이(옹근수)

array_of_requests 요구배렬(손잡이배렬)

출구파라메터

outcount 완료된 요구의 개수(옹근수)

array_of_indices 완료된 연산들의 첨수배렬(옹근수배렬)
array_of_statuses 완료된 연산들의 상태객체배렬(상태배렬)

송신연산들에서 상태

송신연산들에서 상태를 리용하는 경우는 오직 MPI_Test_cancelled함수를 호출하거나 또는 어떤 오유가 있는 경우인데 이 경우에는 상태객체 status의 MPI_ERROR마당이 설정된다.

- MPI_Testall

이전에 시작되였던 모든 통신이 끝났는가를 검사한다.

문법

```
#include "mpi.h"
int MPI_Testall( int count, MPI_Request array_of_requests[],
                         int *flag, MPI_Status array_of_statuses[] )
```

입구파라메터

count 목록길이(옹근수)
array_of_requests 요구배렬(손잡이배렬)

출구파라메터

flag 론리값
array_of_statuses 상태객체배렬(상태배렬)

설명

기발 flag는 오직 모든 요구들이 완료되였을 때에만 참이다. flag가 거짓인 경우에 array_of_requests와 array_of_statuses중 어느것도 갱신되지 않는다.

- MPI_Testany

이미 시작된 통신들중 임의의 하나가 끝났는가를 검사한다.

문법

```
#include "mpi.h"
```

```
int MPI_Testany(int count,  MPI_Request array_of_requests[],
                            int *index, int *flag,  MPI_Status *status )
```

입구파라메터

count 목록길이(옹근수)

array_of_requests 요구배렬(손잡이배렬)

출구파라메터

index 완성된 연산의 첨수, 완성되지 않았으면 MPI_UNDEFINED값(옹근수)

flag 연산중의 하나가 완료되면 참(론리형)

status 상태객체

- MPI_Testsome

일부 지정된 통신들이 끝났는가를 검사한다.

문법

```
#include "mpi.h"
int MPI_Testsome(int incunt, MPI_Request array_of_requests[],
        int *outcount,   int array_of_indices[],  MPI_Status
                                                   array_of_statuses[] )
```

입구파라메터

incount 요구배렬의 길이(옹근수)

array_of_requests 요구배렬(손잡이배렬)

출구파라메터

outcount 완성된 요구개수(옹근수)

array_of_indices 완성된 연산들의 첨수배렬(옹근수배렬)

array_of_statuses 완성된 연산들의 상태객체배렬(상태배렬)

[실례 4.11] MPI_Waitany함수를 리용하는 생산자-소비자

```
typedef struct {
```

```
    char data[MAXSIZE];
    int datasize;
 } Buffer;
Buffer buffer[];
MPI_Status status;
MPI_Request req[];
   …
/* 매 프로쎄스에서 체계의 프로쎄스개수와 자기 번호를 결정한다. */
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
/* 생산자 */
if (rank != size-1) {
    /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(buffer));
    /* 주 순환 */
    while (1) {
        /* 자료파일의 생성과 완충기에 보존된 바이트수의 복귀 */
         produce(buffer->data, &buffer->datasize);
        /* 자료전송 */
         MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
                    size-1, tag, comm);
     }
 }
/* rank=size-1 ; 소비자 */
 else {
   /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(Buffer)*(size-1));
    req=(MPI_Request *)malloc(sizeof(MPI_Request)*(size-1));
    for (i=0; i<= size-1;  i++) {
       /* 매 생산자로부터의 수신초기화 */
      MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
&req[i]);
  /* 기본순환 */
  while (1) {
```

```
    MPI_Waitany(size-1, req, &i, &status);
    /* 실지 수신된 바이트수를 결정 */
    MPI_Get_count(&status, MPI_CHAR, &(buffer[i].datasize));
    /* 소비자함수호출 */
    consume(buffer[i].data, buffer[i].datasize);
     /* 새로운 수신 */
    MPI_Irecv(buffer[i].data,  MAXSIZE,  MPI_CHAR, i, tag,
                comm, &req[i]);
    }
  }
}
```

이 프로그램은 실례 4.7 과 같은 동작을 수행한다. 여기에서 리용하는 MPI_Waitany함수는 끝난 교환연산을 찾기위한 프로그람의 불필요한 중복작업을 없애게 한다. 그러나 이 프로그람은 실례 4.7 의 프로그람과는 달리 생산자의 지연을 방지하지는 못한다.

[실례 4.12] 지연이 없는 생산자봉사프로그람

```
typedef struct {
     char data[MAXSIZE];
     int datasize;
 } Buffer;
Buffer buffer[];
MPI_Status status[];
MPI_Request req[];
int index[];
    …
/* 매 가지는 체계의 콤퓨터수와 자기 번호를 결정한다. */
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
/* 생산자 */
if (rank != size-1) {
/* 완충기의 생성 */
```

```
    buffer=(Buffer *)malloc(sizeof(buffer));
   /* 주 순환 */
    while(1) {
        /* 자료파일의 생성과 완충기에 보존된 바이트수의 복귀 */
         produce(buffer->data, &buffer->datasize);
        /* 자료전송 */
         MPI_Send(buffer->data, buffer->datasize, MPI_CHAR,
                      size-1, tag, comm);
    }
}
/* rank==size-1 ; 소비자 */
else {
  /* 완충기의 생성 */
    buffer=(Buffer *)malloc(sizeof(Buffer)*(size-1));
    req=(MPI_Request *)malloc(sizeof(MPI_Request)*(size-1));
    status=(MPI_Status *)malloc(sizeof(MPI_Status)*(size-1));
    index = (int *)malloc(sizeof(int)*(size-1));
    for (i=0; i<=size-1;  i++)
       /* 매 생산자로부터의 수신 */
      MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
&req[i]);
  /* 주 순환 */
  while (1) {
     MPI_Waitsome(size-1, req, &count, index, &status);
     for (i=0; i<count; i++) {
          j=index[i];
          /* 실지 수신된 바이트수를 결정 */
          MPI_Get_count(&status[i], MPI_CHAR,
                                 &(buffer[j].datasize));
          /* 소비자함수호출 */
          consume(buffer[j].data, buffer[j].datasize);
          /* 새로운 수신 */
          MPI_Irecv(buffer[j].data, MAXSIZE, MPI_CHAR, j, tag,
              comm, &req[j]);
```

```
        }
    }
}
```

이 프로그람은 우의 프로그람에서 나타난 지연문제를 해결한다. 여기서는 소비자가 MPI_Waitany대신에 MPI_Waitsome을 호출한다. 이렇게 함으로써 우선 한번의 호출로써 여러 연산을 수행하므로 통신호출수가 줄어들며 소비자는 임의의 전송을 받으므로 그 무엇도 지연시키지 않는다.

3. 요구객체해방

요구객체는 MPI_Wait함수와 MPI_Test함수를 호출하여 해방할수 있다. 또는 MPI_Request_free함수를 호출하여 해방시킬수도 있다.

- MPI_Request_free
  통신요구객체를 해방한다.

문법

```
#include "mpi.h"
int MPI_Request_free( MPI_Request *request )
```

입구파라메터
  request : 통신요구(손잡이)

4. 통신검색과 무효화

MPI_Probe함수와 MPI_Iprobe함수는 들어오는 통보들을 실제적으로 받지는 않고 검사만을 하도록 한다. 다음에는 응용프로그람이 이 검사정보에 기초하여 통보들을 어떻게 받겠는가를 결정할수 있다. 실례로 응용프로그람은 통보의 길이에 기초하여 수신완충기의 크기를 할당할수 있다.

MPI_Cancel함수는 통신을 취소한다. 가령 프로그람이 비차단송수신연산을 시작하고 후에 이 연산이 어떤 원인으로 완료되지 않을것이라고 판단되면 리용된 기억자원들(송신 및 수신완충기들)을 해방하는데 이 함수를 리용한다.

- MPI_Probe

  통보에 대한 차단검사

문법

```
#include "mpi.h"
int MPI_Probe( int source, int tag, MPI_Comm comm,
                                                MPI_Status *status )
```

입구파라메터

  source 원천프로쎄스번호, 혹은 MPI_ANY_SOURCE(옹근수)

  tag 표적값 혹은 MPI_ANY_TAG(옹근수)

  comm 통신기(손잡이)

출구파라메터

  flag 론리값

  status 상태객체

- MPI_Iprobe

  통보에 대한 비차단검사

문법

```
#include "mpi.h"
int MPI_Iprobe( int source, int tag, MPI_Comm comm, int *flag,
                                                MPI_Status *status )
```

입구파라메터

  source 원천프로쎄스번호, 혹은 MPI_ANY_SOURCE(옹근수)

  tag 표적값 혹은 MPI_ANY_TAG(옹근수)

  comm 통신기(손잡이)

출구파라메터

  flag 론리값

  status 상태객체

[실례 4.13] 수신측에서 통보를 받을 때 차단검사함수의 리용실례

```
MPI_Comm_rank(comm, &rank);
if (rank == 0)
   MPI_Send(I, 1, MPI_INT, 2, 0, comm);
elseif (rank == 1)
   MPI_Send(x, 1, MPI_FLOAT, 2, 0, comm);
elseif (rank == 2) {
    for (k=1; k<=2; k++) {
      MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
       /* 송신자가 프로쎄스 0 이면 한개 옹근수를 수신 */
      if (status.MPI_SOURCE == 0)
      MPI_Recv(I, 1, MPI_INT, 0, 0, comm, &status);
       /* 송신자가 프로쎄스 1 이면 한개 류점수를 수신 */
      else
      MPI_Recv(x, 1, MPI_FLOAT, 1, 0, comm, &status);
   }
}
```

[실례 4.14] 우의 프로그람과 갈지만 문제가 있다.

```
MPI_Comm_rank(comm, &rank);
if (rank == 0)
    MPI_Send(I, 1, MPI_INT, 2, 0, comm);
elseif (rank == 1)
    MPI_Send(x, 1, MPI_FLOAT, 2, 0, comm);
elseif (rank == 2) {
  for (k=1; k<=2; k++) {
      MPI_Probe(MPI_ANY_SOURCE, 0, comm, &status);
      if (status.MPI_SOURCE == 0)
        MPI_Recv(I, 1, MPI_INT, MPI_ANY_SOURCE,
              0, comm,   &status);
      else
        MPI_Recv(x, 1, MPI_FLOAT, MPI_ANY_SOURCE,
```

```
                        0, comm, &status);
    }
}
```

여기서는 수신함수의 송신파라메터값으로 MPI_ANY_SOURCE를 리용하였다. 이때 프로그람의 거동은 각이하다. 송신함수에서 보낸 통보가 아닌 다른 통보를 받을수 있다.

- MPI_Cancel

이미 초기화되였던 통신요구를 취소시킨다.

문법

```
#include "mpi.h"
int MPI_Cancel( MPI_Req   *request )
```

입구파라메터

request : 요구

- MPI_Test_cancelled

요구가 취소되였는가를 검사한다.

문법

```
#include "mpi.h"
int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

입구파라메터

status : 상태객체

출구파라메터

flag : 론리기발

[실례 4.15] MPI_Cancel을 리용한 프로그람

```
MPI_Comm_rank(comm, &rank);
```

```
if (rank == 0)
   MPI_Send(a, 1, MPI_CHAR, 1, tag, comm);
else if (rank == 1) {
        MPI_IRecv(a, 1, MPI_CHAR, 0, tag, comm, &req);
        MPI_Cancel(&req);   /* 수신요구를 취소 */
        MPI_Wait(&req, &status);
        MPI_Test_cancelled(&status, &flag);
        if (flag)   /* 통신취소성공 - 새로 수신 */
        MPI_Recv(a, 1, MPI_CHAR, 0, tag, comm, &status);
}
```

MPI_Cancel은 특이한 경우에만 리용하는 기능으로서 시간을 많이 소비한다.

5. 완충할당과 리용

응용프로그람은 전송되는 통보들의 완충화에 리용될 완충기를 기억기에서 결정하여야 한다. 완충화는 송신프로쎄스에 의하여 수행된다.

- MPI_Buffer_attach

사용자가 정의한 송신용완충기를 붙인다.

문법

```
#include "mpi.h"
int MPI_Buffer_attach( void *buffer, int size )
```

입구파라메터

buffer - 완충기의 시작주소

size - 바이트단위로 계산된 완충기크기

- MPI_Buffer_detach

MPI_Bsend 등에서 리용하기 위하여 현재 완충기를 제거한다.

문법

```
#include "mpi.h"
int MPI_Buffer_detach( void *bufferptr, int *size )
```

출구파라메터

buffer - 완충기의 시작주소
size   - 바이트단위로 계산된 완충기크기

설명

지적자 buffer가 void*로 선언되였다 할지라도 실지로는 void형지적자의 주소이다.

[실례 4.16] 완충기의 리용, 련결, 해제

```
#define BUFFSIZE 10000
int size;
char *buff;
buf= (char *)malloc(BUFFSIZE);
MPI_Buffer_attach(buff, BUFFSIZE);
  /* MPI_Bsend함수는 10 000 byte크기의 완충기를 리용할수 있다.*/
   …
MPI_Buffer_detach(&buff, &size);
  /* 완충기의 크기는 0 byte로 된다. */
  …
MPI_Buffer_attach(buff, BUFFSIZE);
 /* 완충기에 다시 10 000 byte를 할당 */
  …
```

6. 동시호출가능성

MPI에서 매개 프로세스는 마치도 병렬로 실행되는것처럼 보이는 몇개의 부분프로쎄스(제각기 입구자료 및 중간자료와 상태벡토르를 가지고 일정한 기억구역에 들어있는 프로그람코드)들을 구성성분으로 가질수도 있다.

매개 부분프로쎄스에는 실행시간구간이 순서대로 주어진다. 이 시간구간이 끝나는데 따라 현재의 부분프로쎄스는 중단되고 대기렬에 들어가며 조종은 다음 순서의 부분프로쎄스에 일정한 시간동안 넘어간다. MPI는 다중부분프로쎄스실행에서 차단통신함수의 호출과 부분프로쎄스계획화와의 호상작용을 처리하지 못한다. 부분프로쎄스가 차단통신을 진행하는 경우 송신완충기에 대한 접근은 이 부분프로쎄스에 대해서만 차단된다. 부분프로쎄스들이 공통적인 자료를 가지고있을수 있기때문에 다음과 같은 상황이 발생할수 있다. 현재의 부분프로쎄스는 어떤 완충기의 송신처리를 시작하였지만 시간구간이 끝났으므로 중단되며 조종은 다음 부분프로쎄스에 넘어간다. 이때 이전 부분프로쎄스의 송신완충기에 접근할수 있다. 이 경우 이 완충기접근은 다른 가지에 의해 차단되지 않는다. 결국 완충기의 송신결과도 이전의 부분프로쎄스에 의해 결정되지 않는다.

7. 송수신결합함수

한번 호출하여 송신과 수신을 함께 하는 특수한 함수이다. 이 함수는 프로쎄스가 통보를 보내고 받는데 유익하다. 실례로 두 프로쎄스사이의 자료교환이나 프로쎄스사슬에서의 자료이동 등을 들수 있다.

- MPI_Sendrecv
통보를 수신하고 보낸다.

문법
```
#include "mpi.h"
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
                  sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag, MPI_Comm comm, MPI_Status
                 *status )
```

입구파라메터

sendbuf	송신완충기의 시작주소
sendcount	송신완충기의 요소개수(옹근수)

sendtype　송신완충기의 요소자료형(손잡이)
dest　목적프로쎄스의 번호(옹근수)
sendtag　송신통보표적(옹근수)
recvcount　수신완충기의 요소개수(옹근수)
recvtype　수신완충기의 요소자료형(손잡이)
source　원천프로쎄스의 번호(옹근수)
recvtag　수신통보표적(옹근수)
comm　통신기(손잡이)

출구파라메터
recvbuf　출구완충기의 시작주소
status　상태객체, 수신연산과 관련

- MPI_Sendrecv_replace

한개의 완충기를 리용하여 송신하고 수신한다.

문법

```
#include "mpi.h"
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
        datatype, int dest, int sendtag, int source, int recvtag,
                    MPI_Comm comm, MPI_Status *status )
```

입구파라메터
count　송신 및 수신완충기에서 요소수(옹근수)
datatype　송신 및 수신완충기에서 요소자료형(옹근수)
dest　목적프로쎄스의 번호(옹근수)
sendtag　송신통보표적(옹근수)
source　원천프로쎄스의 번호(옹근수)
recvtag　수신통보표적(옹근수)
comm　통신기(손잡이)

출구파라메터
buf　송신 및 수신완충기의 시작주소

status 상태객체

8. 고정통신호출

같은 파라메터목록을 가진 통신을 병렬계산의 내부순환에서 여러번 진행해야 할 필요가 자주 제기된다. 이 경우에 고정통신호출에 파라메터목록을 한번만 보내고 다음에는 송신을 시작하고 끝내도록 다중호출을 써서 통신을 최량화하는것이 가능하다.

- MPI_Send_init

표준전송에 대한 손잡이를 구축

문법

```
#include "mpi.h"
int MPI_Send_init(void *buf, int count, MPI_Datatype datatype,
      int dest, int tag, MPI_Comm comm, MPI_Request *request )
```

입구파라메터

buf	송신완충기의 시작주소(선택)
count	송신하는 요소의 개수(옹근수)
datatype	매 요소의 자료형(손잡이)
dest	목적프로쎄스의 번호(옹근수)
tag	통보표적(옹근수)
comm	통신기(손잡이)

출구파라메터

request 통신요구(손잡이)

- MPI_Recv_init

수신을 위한 손잡이를 구축

문법

```
#include "mpi.h"
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype,
```

int source, int tag, MPI_Comm comm, MPI_Request *request)

입구파라메터

buf 수신완충기의 시작주소

count 수신되는 요소수(옹근수)

datatype 매 요소형(손잡이)

source 원천프로세스의 번호 혹은 MPI_ANY_SOURCE(옹근수)

tag 통보표적 혹은 MPI_ANY_TAG(옹근수)

comm 통신기(손잡이)

출구파라메터

request : 통신요구(손잡이)

- MPI_Start

요구손잡이를 가지고 통신을 시작하도록 한다.

문법

```
#include "mpi.h"
int MPI_Start( MPI_Request *request)
```

입구파라메터

request : 통신요구(손잡이)

- MPI_Startall

요구들의 모으기를 시작한다.

문법

```
#include "mpi.h"
int MPI_Startall(int count, MPI_Request array_of_requests[] )
```

입구파라메터

count 목록 길이(옹근수)

array_of_requests 요구배렬(손잡이배렬)

- MPI_Ssend_init
  동기전송을 위한 손잡이를 구축한다.

문법

  #include "mpi.h"
  int MPI_Ssend_init(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

입구파라메터

buf 송신완충기의 시작주소(선택)
count 전송할 요소개수(옹근수)
datatype 매 요소의 자료형(손잡이)
dest 목적프로쎄스의 번호(옹근수)
tag 통보표적(옹근수)
comm 통신기(손잡이)

출구파라메터

  request : 통신요구(손잡이)

- MPI_Bsend_init
  완충전송을 위한 손잡이를 구축한다.

문법

  #include "mpi.h"
  int MPI_Bsend_init(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

입구파라메터

buf - 송신완충기의 시작주소
count - 전송할 요소의 수
datatype - 매 요소의 형
dest - 목적프로쎄스번호
tag - 통보표적
comm - 통신기

출구파라메터

request - 통신요구

- MPI_Rsend_init

기다림전송에 대한 손잡이를 구축한다.

문법

```
#include "mpi.h"
int MPI_Rsend_init( void *buf, int count, MPI_Datatype
                    datatype, int dest, int tag, MPI_Comm comm,
                                          MPI_Request *request )
```

입구파라메터

buf 송신완충기의 시작주소(선택)

count 전송할 요소개수(옹근수)

datatype 매 요소형(손잡이)

dest 목적프로쎄스의 번호(옹근수)

tag 통보표적(옹근수)

comm 통신기(손잡이)

출구파라메터

request : 통신요구(손잡이)

제 5 절 집합연산

1. 개념

집합통신은 함수의 내부통신기파라메터로 지적해준 그룹안의 모든 프로쎄스들에 관해서 자료교환을 보장한다. 앞으로는 그룹을 내부통신기에 련결된 프로쎄스부분모임으로 본다. MPI는 다음과 같은 집합통신함수들을

제공하고있다.

- 동기화(barrier)
  그룹의 모든 프로쎄스들을 동기화한다.
- 대역통신함수들
  *그룹의 한 프로쎄스로부터 그룹의 전체 프로쎄스들에로의 자료전송 (broadcast)
  *그룹의 전체 프로쎄스들로부터 이 그룹의 한 프로쎄스에로의 자료수집(gather)
  *그룹의 한 프로쎄스로부터 그룹의 모든 프로쎄스들에로 자료분산 (scatter)
  *그룹의 모든 프로쎄스에 따르는 순환에서의 자료수집(allgather)
  *그룹의 모든 성원들로부터 모두에로의 자료분산/수집 (scatter/gather)
- sum, max, min 형 대역축소연산 혹은 사용자정의함수들
  * 결과가 그룹의 모든 프로쎄스들에 보내지는 축소와 결과가 그룹의 한 프로쎄스에만 보내지는 축소(allreduce/reduce)
  * 련결축소, scatter연산, 그룹의 모든 원소들에 따르는 중첩

집합함수들의 문법과 의미는 점대점(point-to-point)연산과 어느 정도 류사하지만 비교적 제한적이다. 한가지 제한은 점대점(point-to-point) 연산과는 달리 전송한 자료수는 수신자가 지적한 자료수와 정확히 일치하여야 한다는것이다.

집합함수들에서 기본적으로 같은 측면은 이것들이 모두 차단이라는것이다. 집합함수들은 표적파라메터를 리용하지 않으며 매 그룹내에서 집합호출통신구역들은 실행순서에 준하여 엄격하게 규정되여있다.

또 한가지 특징은 집합연산들이 모두 자료교환수법으로서 점대점연산의 표준규칙과 류사한 수법을 리용한다는것이다.

2. 동기화와 자료의 방송

MPI_Barrier함수는 그룹안의 모든 프로쎄스성원들이 모두 이 함수를 호출할 때까지 그 실행을 차단시킨다.

- MPI_Barrier

모든 프로세스들이 이 함수에 도달할 때까지 차단시킨다.

문법

```
#include "mpi.h"
int MPI_Barrier( MPI_Comm comm )
```

입구파라메터

comm - 통신기

- MPI_Bcast

번호가 root인 프로세스로부터 그룹의 다른 모든 프로세스들에로 통보를 방송한다.

문법

```
#include "mpi.h"
int MPI_Bcast( void *buffer, int count, MPI_Datatype
 datatype, int root, MPI_Comm comm )
```

입구파라메터

buffer - 완충기의 시작주소
count - 완충기의 요소의 수
datatype - 완충기요소의 자료형
root - 방송하는 프로세스의 번호
comm - 통신기

알고리듬

선형알고리듬을 리용하여 하나의 블로크안에 있는 자료를 첫번째 프로세스로부터 다른 모든 프로세스들에로 방송한다.

[실례 5.1] MPI_Bcast리용실례

```
MPI_Comm comm;
int array[100];
```

```
int root=0;
  …
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

0번 프로세스의 옹근수배렬 array안에 있는 100개의 int형 옹근수들을 0번 프로세스로부터 그룹의 모든 프로세스들에 전송한다.

3. 자료수집

- MPI_Gather

그룹안의 모든 프로세스들로부터 지정된 완충기의 내용을 프로세스 root의 보다 큰 다른 완충기에 모아놓는다.

문법

```
#include "mpi.h"
int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype
 sendtype, void *recvbuf, int recvcount, MPI_Datatype
                        recvtype, int root, MPI_Comm comm )
```

입구파라메터

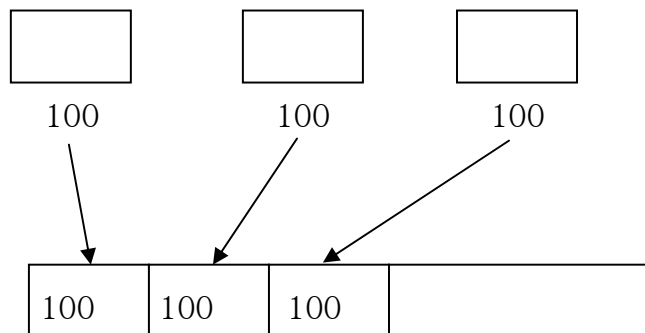
sendbuf 송신완충기의 시작주소(선택)
sendcount 송신완충기의 요소의 개수(옹근수형)
sendtype 송신완충기요소의 자료형(손잡이)
recvcount 매 프로세스에서 수신요소들의 수(옹근수)
recvtype 수신완충기요소의 자료형(손잡이)
root 수신프로세스의 번호(옹근수)
comm 통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 시작주소

[실례 5.2] MPI_Gather함수의 리용실례를 보여준다.

```
MPI_Comm comm;
```

```
int gsize, sendarray[100];
int root, *rbuf;

MPI_Comm_size(comm, &gsize);
rbuf= (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT,
                root, comm);
```

[실례 5.3] 우의 코드와 비슷하지만 여기에서는 뿌리프로세스만이 수신용 완충기(rbuf)를 할당한다.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, myrank;

MPI_Comm_rank(comm, &myrank);
if (myrank == root){
    MPI_Comm_size(comm, &gsize);
    rbuf= (int *)malloc(gsize*100*sizeof(int));
 }
```

```
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT,
             root, comm);
```

그룹의 매 프로세스로부터 100 개의 int형 옹근수를 root의 수신완충기 rbuf에 모아놓는다.

[실례 5.4] 우의 코드와 비슷하지만 여기서는 임의의 datatype를 리용한다.

```
MPI_Comm comm;
int gsize ;
int *rbuf;
int sendarray[100];
int root=0;
…
MPI_Comm_size(comm, &gsize);
rbuf= (int *)malloc(gsize*100*sizeof(int));

MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT,
             root, comm);
```

다음으로 MPI_Gather함수의 변종인 MPI_Gatherv함수를 보기로 한다.

- MPI_Gatherv

그룹안의 모든 프로세스들로부터 완충기의 내용을 모든 프로세스들의 지정된 위치에로 수집한다. MPI_Gather와의 차이점은 모든 프로세스들에서 보내고 받는 자료의 개수가 다 차이난다는것이다.

문법

```
#include "mpi.h"
int MPI_Gatherv( void *sendbuf, int sendcount, MPI_Datatype
          sendtype,   void *recvbuf, int *recvcounts, int *displs,
           MPI_Datatype recvtype, int root, MPI_Comm comm )
```

입구파라메터

sendbuf	송신완충기의 시작주소(선택)
sendcount	송신완충기에서 요소의 개수(옹근수)
sendtype	송신완충기요소의 자료형(손잡이)
recvcounts	매 프로쎄스로부터 수신되는 요소수를 담고있는 옹근수배렬(크기는 그룹크기)(root에서만 의미가 있음)
displs	그룹크기만한 옹근수배렬, 요소 i는 프로쎄스 i로부터 들어오는 자료를 넣을 recvbuf의 시작으로부터의 변위
recvtype	수신완충기요소의 자료형 (손잡이)
root	수신프로쎄스의 번호(옹근수)
comm	통신기(손잡이)

출구파라메터

recvbuf : 수신된 완충기의 시작주소

[실례 5.5] MPI_Gatherv를 리용한 실례

```
MPI_Comm comm;
int gsize, rank, sendarray[100];
int   root, *rbuf;
int   displs, i, rcounts;
int   stride=100;

MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsizestride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0;i<gsize;++i) {
     displs[i]=i*stride;
     rcounts[i]=100;
}
MPI_Gatherv(sendarray, 100 , MPI_INT, rbuf, rcounts, displs,
     MPI_INT, root, comm);
```

매 프로쎄스는 100 개의 원소를 전송한다. 수신프로쎄스에서 이 블로크

들은 수신완충기시작으로부터 어떤 변위만큼 떨어지면서 할당되여야 한다. 이것을 위해 MPI_Gatherv함수와 displs파라메터를 리용한다. 우에서 stride는 변위와 변위사이의 폭을 의미하는데 100으로 한다.

[실례 5.6] MPI_Gatherv를 리용한 실례

우의 실례와 같지만 100*150 행렬에서 0번째 렬의 100개 원소를 전송한다.

```
  …
MPI_Comm comm;
int gsize, sendarray[100][150];
int   root, *rbuf, stride;
int   displs, i, rcounts;
MPI_Datatype stype;
…
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsizestride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0;i<gsize;++i)
   { displs[i]=i*stride;
     rcounts[i]=100;
   }
/* 100*150 행렬에서 한개 렬에 대한 자료형구축 */
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
MPI_Gatherv(sendarray, 1, &stype, rbuf, rcounts, displs,
     MPI_INT, root, comm);
```

[실례 5.7] 프로쎄스 i는 100*150 행렬의 i번째 렬의 (100-i)개 원소를 전송한다.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int   root, *rbuf, stride, myrank;
```

```
int   displs, i, rcounts;
MPI_Datatype stype;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsizestride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0;i<gsize;++i) {
      displs[i]=i*stride;
      rcounts[i]=100-i;
}
/* 전송하는 한개 렬에 대한 자료형 구축 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr 는 100*150 행렬에서 myrank번째 렬의 시작주소 */
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs,
      MPI_INT, root, comm);
```

결국 매 프로세스로부터 각이한 개수의 자료들을 받게 된다.

4. 자료분산

자료분산에는 MPI_Scatter함수와 MPI_Scatterv함수들이 있다.

- MPI_Scatter

자료를 그룹의 한 프로세스로부터 다른 프로세스로 보낸다.

문법

```
#include "mpi.h"
int MPI_Scatter( void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,
                                                   int root, MPI_Comm comm )
```

입구파라메터

sendbuf   완충기주소
sendcount 매 프로쎄스에 전송될 요소의 수(옹근수)
sendtype 송신완충기원소들의 자료형(손잡이)
recvcount 수신완충기의 요소수(옹근수)
recvtype 수신완충기요소들의 자료형(손잡이)
root   전송프로쎄스의 번호(옹근수)
comm   통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 주소

[실례 5.8] MPI_Scatter의 리용실례

```
MPI_Comm comm;
int gsize, *sendbuf;
int rbuf[100];
int root=0;
...
MPI_Comm_size(comm, &gsize);
sendbuf=(int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100,
                        MPI_INT, root, comm);
```

프로쎄스 root는 sendbuf안의 int형 옹근수 100 개를 그룹의 모든 프로쎄스들에 전송하며 완충기 rbuf에 넣는다.

- MPI_Scatterv

매 프로쎄스에 불규칙적인 개수의 자료들을 전송한다.

문법

```
#include "mpi.h"
int MPI_Scatterv( void *sendbuf,  int *sendcounts, int *displs,
```

```
MPI_Datatype sendtype,    void *recvbuf, int recvcount,
MPI_Datatype recvtype,   int root,   MPI_Comm comm )
```

입구파라메터

sendbuf	송신완충기의 시작주소
sendcounts	매 프로쎄스에 보낼 요소수를 지정하는 옹근수배렬 (크기는 그룹크기)
displs	옹근수배렬(그룹크기), i째 요소는 sendbuf로부터의 변위값으로서 프로쎄스 i로 보낼 블로크의 시작위치를 지정
sendtype	송신완충기요소들의 자료형(손잡이)
recvcount	수신완충기의 요소수(옹근수)
recvtype	송신완충기요소들의 자료형(손잡이)
root	전송프로쎄스의 번호(옹근수)
comm	통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 주소

[실례 5.9] MPI_Scatterv의 리용실례 1, [실례 5.5]와 반대이다.

뿌리프로쎄스는 다른 프로쎄스들에 100 개의 요소를 전송하는데 이 100 개의 요소들은 송신완충기에 걸음 stride(>100)만큼씩 떨어져서 놓이게 된다.

```
MPI_Comm comm;
int gsize, *sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size(comm, &gsize);
sendbuf=(int *)malloc(gsize*stride*sizeof(int));
...
displs=(int *)malloc(gsize*sizeof(int));
scounts=(int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i]=i*stride;
    scounts[i]=100;
```

```
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100,
                              MPI_INT, root, comm);
```

[실례 5.10] MPI_Scatterv의 리용실례 2

프로쎄스 root의 완충기에서 전송블로크들은 블로크들사이에 일정한 걸음씩 떨어져있는데 이 걸음크기가 가변적이다. 수신측인 프로쎄스 i에서 100*150 행렬의 i번째 렬에로 100-i개의 요소들을 받는다.

```
MPI_Comm comm;
int gsize, recvarray[100][150], *rptr;
int root, sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
…
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride=(int *)malloc(gsize*sizeof(int));
…
/* i=0부터 i=gsize-1까지의 stride[i]를 설정한다. */
…
displs=(int *)malloc(gsize*sizeof(int));
scounts=(int *)malloc(gsize*sizeof(int));
offset=0;
for (i=0; i<gsize; ++i) {
    displs[i]=offset;
    offset+=stride[i];
    scounts[i]=100-i;
}
/* 수신하는 렬을 위한 자료형구축 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr=&recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
```

```
                 root, comm);
```

- MPI_Allgather

자료를 모든 과제들로부터 받아 모두에게로 분산시킨다.

문법

```
#include "mpi.h"
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype
    sendtype, void *recvbuf, int recvcount, MPI_Datatype
    recvtype, MPI_Comm comm )
```

입구파라메터

sendbuf	송신완충기의 시작주소
sendcount	송신완충기의 요소수(옹근수)
sendtype	송신완충기요소들의 자료형(손잡이)
recvcount	어떤 프로쎄스로부터의 수신되는 요소수(옹근수)
recvtype	수신완충기요소들의 자료형(손잡이)
comm	통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 주소

[실례 5.11] MPI_Allgather의 리용실례

그룹의 매 프로쎄스에서는 모든 프로쎄스부터 100개 요소씩 받는다.

```
MPI_Comm comm;
int gsize, sendarray[100];
int *rbuf;

    ...
MPI_Comm_size(comm, &gsize);
rbuf=(int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf,
                                            100, MPI_INT, comm);
...
```

- MPI_Allgatherv

모든 과제들로부터 자료를 받아 그것을 모두에게로 전송한다.

MPI_Gatherv와 다른 점은 뿌리프로세스 하나가 아닌 모든 프로세스들이 자료를 받는다는것이다.

문법

```
#include "mpi.h"
int MPI_Allgatherv(void *sendbuf, int sendcount , MPI_Datatype
 sendtype, void *recvbuf, int *recvcounts, int *displs,
              MPI_Datatype recvtype, MPI_Comm comm )
```

입구파라메터

sendbuf	송신완충기의 시작주소
sendcount	송신완충기의 요소수(옹근수)
sendtype	송신완충기요소들의 자료형(손잡이)
recvcounts	매 프로세스로부터 받을 요소수가 들어있는 옹근수배렬(크기는 그룹크기)
displs	옹근수배렬(크기는 그룹크기), i는 프로세스 i로부터 들어오는 자료가 놓일 시작위치(recvbuf로부터의 변위)
recvtype	수신완충기요소들의 자료형(손잡이)
comm	통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 주소

5. 분산/수집

- MPI_Alltoall

전체로부터 전체 프로세스에로 자료를 보낸다.

문법

```
#include "mpi.h"
int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype
 sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,
                                            MPI_Comm comm )
```

입구파라메터

sendbuf - 송신완충기의 시작주소

sendcount - 매 프로세스에로 보내는 요소의 수
sendtype - 보내는 완충기요소의 자료형
recvcount - 임의의 처리로부터 수신되는 요소의 수
recvtype - 수신되는 완충기요소의 자료형
comm - 통신기

출구파라메터
recvbuf - 수신완충기의 주소

- MPI_Alltoallv
전체로부터 전체 프로세스에로 가변크기의 자료를 보낸다.

문법
#include "mpi.h"
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

입구파라메터
sendbuf - 송신완충기의 시작주소
sendcounts - 매 프로쎄스로 송신요소들의 개수를 서술하는 옹근수배렬(그룹크기)
sdispls - 옹근수배렬(그룹크기)
sendtype - 보내는 완충기요소의 자료형
recvcounts - 모든 처리기들로부터 수신될수 있는 요소들의 최대수를 서술하는 옹근수배렬(그룹크기)
rdispls - 옹근수배렬(그룹크기)
recvtype - 수신완충기요소의 자료형
comm - 통신기

출구파라메터
recvbuf - 수신완충기의 주소

6. 대역축소연산

여기서는 그룹의 모든 성원들에서의 대역축소연산(sum, max, 론리적 AND 등)들을 고찰한다. 축소연산은 미리 정의된 연산들중의 하나일수도 있고 사용자가 정의하는 연산일수도 있다. 대역축소연산들에는 한개 마디에 결과를 보내는 축소와 결과를 모든 마디들에 다 보내는 축소가 있다. 이 외에 Reduce_scatter와 같이 축소와 분산을 결합한 연산도 있다.

먼저 미리 정의된 연산들을 보기로 한다.

MPI_MAX - 최대
MPI_MIN - 최소
MPI_SUM - 합
MPI_PROD - 적
MPI_LAND - 론리적
MPI_BAND - 비트론리적
MPI_LOR - 론리합
MPI_BOR - 비트론리합
MPI_LXOR - 배타론리합
MPI_BXOR - 배타적비트론리합
MPI_MAXLOG - 최대값과 그 위치
MPI_MINLOG - 최소값과 그 위치

- MPI_Reduce

모든 프로쎄스들에 있는 값들을 한개 프로쎄스의 단일값으로 축소한다.

문법

```
#include "mpi.h"
int MPI_Reduce( void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
                                                  comm )
```

입구파라메터

sendbuf 송신완충기의 주소
count 송신완충기에 있는 요소개수(옹근수)

datatype　송신완충기의 요소들의 형(손잡이)
op　축소연산(손잡이)
root　결과를 받을 프로세스의 번호(옹근수)
comm　통신기(손잡이)

출구파라메터
recvbuf : 수신완충기의 주소

[실례 5.12] PAR_BLAS1 은 그룹의 프로세스들에 들어있는 두 벡토르의 스칼라적을 계산하여 결과를 0 번 프로세스에 주는 연산자이다.

```
PAR_BLAS1(m, a, b, c, comm)
int m;
float a[], b[];    /* 배렬들의 국부적인 대역 */
float c;   /* 결과 */
MPI_Comm comm;
{
   int i;
   float sum;
   /* 국부합 */
   sum=0.0;
   for (i=0;i<m; i++)
      sum=sum+a[i]*b[i];
   /* 대역합 */
  MPI_Reduce(&sum, &c, 1, MPI_FLOAT, MPI_SUM, 0, comm);
  return 0;
}
```

[실례 5.13] PAR_BLAS2 는 그룹의 프로세스들에 들어있는 벡토르와 행렬의 적을 계산하여 결과를 0 번 프로세스에 주는 연산자이다.

```
PAR_BLAS2(m, n, a, b, c, comm)
int m, n;
```

```
float a[], b[][n]; /* 배열들의 국부적인 대역 */
float c[n];  /* 결과 */
MPI_Comm comm;
{
 int i, j;
 float sum[n];
 /* 국부합 */
 for (j=0;j<n; j++) {
  sum[j]=0.0;
  for (i=0; i<m; i++)
     sum[j]=sum[j]+a[i]*b[i][j];
 }
 /* 대역적인 합을 구하고 결과를 0번 프로세스에로 보내기 */
 MPI_Reduce(sum, c, n, MPI_FLOAT, MPI_SUM, 0, comm);
 return 0;
}
```

- MPI_Allreduce

모든 프로세스들로부터 값들을 집합연산하여 그 결과를 모든 프로세스들에 전송한다.

문법

```
#include "mpi.h"
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

입구파라메터

- sendbuf - 송신완충기의 시작주소
- count - 송신완충기의 요소의 수
- datatype - 송신완충기의 요소의 자료형
- op - 연산
- comm - 통신기

출구파라메터

recvbuf - 수신완충기의 시작주소

[실례 5.14] PAR_BLAS2 는 그룹의 모든 프로세스들에 들어있는 벡토르와 행렬의 적을 계산하여 결과를 모든 프로세스들에 주는 연산자이다.

```
PAR_BLAS2(m, n, a, b, c, comm)
int m, n;
float a[], b[][n]; /* 배렬들의 국부적인 대역 */
float c[n];  /* 결과 */
MPI_Comm comm;
{
  int i, j;
  float sum[n];

  /* 국부합 */
  for (j=0;j<n; j++) {
    sum[j]=0.0;
    for (i=0;i<m;i++)
     sum[j]=sum[j]+a[i]*b[i][j];
  }
 /* 대역적인 합을 구하고 결과를 모든 프로세스들에로 보내기 */
 MPI_Allreduce(sum, c, n, MPI_FLOAT, MPI_SUM, comm);
 return 0;
}
```

- MPI_Reduce_scatter

값을 집합축소하고 그 결과를 모두에게 전송한다.

문법

```
#include "mpi.h"
int MPI_Reduce_scatter( void *sendbuf, void *recvbuf, int
 *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm
        comm )
```

입구파라메터

sendbuf 송신완충기의 시작주소

recvcounts 매 프로쎄스에 분산되는 결과의 요소수를 지정하는 옹근수 배렬, 배렬은 모든 프로쎄스들에서 같아야 한다.

datatype 입구완충기의 요소들의 자료형(손잡이)

op 연산(손잡이)

comm 통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 시작주소

[실례 5.15] PAR_BLAS3 은 그룹의 모든 프로쎄스들에 분산되여있는 벡토르에 행렬을 곱하여 결과를 분할된 행렬에로 보내주는 연산자이다.

```
PAR_BLAS3(m, n, k, a, b, c, comm)
int m, n, k;
float a[], b[][]; /* 배렬들의 국부적인 대역 */
float c[];   /* 결과 */
MPI_Comm comm;
{
 int i, j, gsize;
 float sum[n];
 int *recvbuf;

 /* 배렬대역크기를 모든 프로쎄스들에 보내기 */
 MPI_Comm_size(comm, &gsize);
 recvbuf= malloc(gsize*sizeof(int));
 MPI_Allgather(&k, 1, MPI_INT, recvbuf, 1, MPI_INT, comm);
 /* 국부합 */
 for (j=0;j<n; j++) {
  sum[j]=0.0;
  for (i=0;i<m;i++)
      sum[j]=sum[j]+a[i]*b[i][j];
 }
```

```
  /* 대역합과 벡토르 c의 분할 */
  MPI_Reduce_scatter(sum, c, recvbuf, MPI_FLOAT,
                                            MPI_SUM, comm);
  /* 분할된 벡토르에 결과 보내기 */
return 0;
}
```

- MPI_Scan

그룹에 있는 자료의 부분적인 축소를 진행한다. 프로세스 i 의 수신완충기에 프로세스 0~i 까지의 송신완충기값들이 집합축소된다.

문법

```
#include "mpi.h"
int MPI_Scan( void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

입구파라메터

sendbuf	송신완충기의 시작주소
count	입구완충기의 요소수(옹근수)
datatype	입구완충기의 요소자료형(손잡이)
op	연산(손잡이)
comm	통신기(손잡이)

출구파라메터

recvbuf : 수신완충기의 시작주소

7. 축소를 위한 사용자정의연산들

- MPI_Op_create

사용자가 정의한 집합함수손잡이를 창조한다.

문법

```
#include "mpi.h"
```

```
int MPI_Op_create(MPI_User_function *function, int commute,
                                        MPI_Op *op )
```

입구파라메터

function 사용자정의함수(함수)

commute 가환이면 참, 그렇지 않으면 거짓

출구파라메터

op : 연산(손잡이)

- MPI_Op_free

사용자가 정의한 집합함수손잡이를 해방한다.

문법

```
#include "mpi.h"
int MPI_Op_free( MPI_Op *op )
```

입구파라메터

op : 연산(손잡이)

설명

op는 MPI_OP_NULL로 설정된다.

[실례 5.16] 복소수배렬들의 적을 계산

```
typedef struct {
 double real, imag;
} Complex;

/* 사용자정의함수 */
void myProd(Complex in, Complex inout, int *len,
                MPI_Datatype *dptr)
{
  int i;
  Complex c;
```

```
  for (i=0; i< *len; ++i) {
      c.real= inout->real*in->real-inout->imag*in->imag;
      c.imag= inout->real*in->imag-inout->imag*in->real;
      *inout=c;
      in++;
      inout++;
  }
}

/* 함수의 호출 */
 ...
/* 매 프로쎄스는 크기가 100인 복소수배렬을 가진다.*/
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;
/* 복소수형구축 */
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
/* 복소수들을 곱하는 사용자연산을 생성*/
MPI_Op_create(myProd, true, &myOp);
MPI_Reduce(a, answer,100, ctype, myOp, root, comm);
  /* 프로쎄스 root에 100개의 복소수를 곱한 결과가 들어간다.*/
```

제 6 절 사용자정의자료형과 자료의 묶기

1. 개념

우에서 본 MPI의 통신기구들은 기억기에서 련속적으로 놓인 같은 형의 요소렬들에 대한 송수신만을 허용한다. 그러나 실천적으로는 구조체형 혹은 배렬구조체와 같은 동종이 아닌 요소렬들을 전송하여야 할 필요성이 많이 제기된다.

MPI는 이러한 문제를 해결하기 위한 다음과 같은 두가지 방법들을

제공하고있다.

첫째로, 사용자는 임의의 자료형을 지정할수 있으며 이 형을 MPI통신함수들에서 미리 정의된 기초형들과 류사하게 리용한다.

둘째로, 전송프로세스는 몇개의 불련속기억구역들에 있는 자료들을 하나의 련속완충기에 묶기하여 그것을 전송할수 있으며 수신프로쎄스는 완충기에 얻어진 자료를 갈라서 몇개의 비겹침기억구역들에 보관할수 있다.

모든 MPI통신함수들은 자료형파라메터인 datatype를 가지고있다. 가장 단순한 경우에 이것은 옹근수(int) 혹은 류점수(float)와 같은 기초자료형이다.

MPI에서는 기초자료형들이 리용되는 모든 곳에서 사용자정의자료형들을 쓸수 있다. 이것들은 프로그람작성언어의 의미에서 "형"이 아니며 다만 기초형들의 기억기배치를 서술하는 형구축함수들에 의하여 MPI가 만드는 "형" 일 뿐이다.

MPI는 사용자정의형들을 통하여 기초자료형들을 조합한 구조체와 같은 종합적인 자료교환을 보장한다.

△ 사용자자료형은 다음의 특징을 가진다.

-기초형들의 렬이다.
-시작주소로부터 시작한 이 형들의 옹근수변위(바이트단위)렬이다.

변위가 반드시 정의 값이고 증가순서로 될 필요는 없다. 결국 사용자정의형목록에서 형값들의 순서는 기억기에서의 그것들의 순서와 반드시 일치하지 않아도 되며 형의 값들은 목록에서 한번이상 나타날수 있다. 이러한 쌍의 렬들을 형의 넘기기라고 한다.

사용자정의형은 송신 및 수신함수에서 기초형파라메터와 같이 형파라메터로 리용될수 있다.

실례로 함수 MPI_Send(buf, 1, datatype, …)은 사용자정의형의 시작주소 buf를 송신완충기의 주소로 리용하며 전송자료의 형은 파라메터 datatype를 리용한다.

[실례 6.1]

Type={(double, 0), (char, 8)}(double의 변위는 0 이고 char의 변위는 8)라고 하자. 그리고 이밖에 실수(double)는 8byte의 배수로 되는 주소에

따라 놓인다고 하자. 이때 형의 아래한계 lb(Type)=min(dispj)=0, 웃한계 ub(Type)=max(dispj+sizeof(typej))+e=(8+1+7)=16 이고 이 형의 대역은 16(8byte(double)+1byte(char)=9 로서 8 의 배수로 되는 다음수는 16)이다. 이 자료형의 배치는 아래의 그림과 같다.

double	char	
8byte	1byte	e=7byte

자료형에 관한 정보를 주는 함수들은 다음과 같다.

- MPI_Type_extent

자료형의 범위를 준다.

문법

```
#include "mpi.h"
int MPI_Type_extent( MPI_Datatype datatype, MPI_Aint *extent )
```

입구파라메터

datatype : 자료형(손잡이)

출구파라메터

extent : 자료형범위(옹근수)

- MPI_Type_size

사용자정의형에서 항목들이 차지한 크기(바이트단위)를 준다.

문법

```
#include "mpi.h"
int MPI_Type_size( MPI_Datatype datatype, int *size )
```

입구파라메터

datatype : 자료형(손잡이)

출구파라메터

size : 자료형크기(옹근수)

[실례 6.2] datatype가 실례 6.1 에서 결정한 형 Type를 취한다고 하자.

이때 MPI_Type_extent(datatype, i)는 i=16을 주며 MPI_Type_size (datatype, i)를 호출하면 i=9(8byte(double) + 1byte(char) = 9)로 된다.

- MPI_Type_lb

자료형의 아래한계를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Type_lb( MPI_Datatype datatype, MPI_Aint
                                        *displacement )
```

입구파라메터

datatype : 자료형(손잡이)

출구파라메터

displacement : 원천으로부터 아래한계의 변위(바이트단위)(옹근수)

- MPI_Type_ub

자료형의 웃한계값을 돌려준다.

문법

```
#include "mpi.h"
int MPI_Type_ub( MPI_Datatype datatype, MPI_Aint
                                        *displacement )
```

입구파라메터

datatype : 자료형(손잡이)

출구파라메터

displacement : 원천으로부터 웃한계의 변위(바이트단위)(옹근수)

2. 형구축자

2.1 련속자료형구축자

- MPI_Type_contiguous

련속적인 자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_contiguous( int count, MPI_Datatype
                    old_type, MPI_Datatype *newtype)
```

입구파라메터

count　복사개수(부아닌 옹근수)

oldtype 낡은 자료형(손잡이)

출구파라메터

newtype : 새 자료형(손잡이)

[실례 6.3] oldtypes가 대역 18 count=3인 형 {(double, 0), (char, 8)}의 넘기기를 취한다고 하자. newtype에 의하여 복귀한 datatype형의 넘기기는 {(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)}이다. 즉 double 및 char요소들은 변위 0, 8, 16, 24, 32, 40으로 련이어 놓인다. 일반적으로 대역이 ex인 형넘기기

oldtype={(type0, disp0),…(typen-1, dispn-1)}

이 있다고 하자. 이때 newtype는 다음과 같이 결정되는 count*n개의 요소로 이루어진 형넘기기를 취한다.

{(type0, disp0), …(typen-1, dispn-1), (type0, disp0+ex), …(typen-1, dispn-1+ex), …(type0, disp0+ex*(count-1)), …, (typen-1, dispn-1+ex*(count-1))}

2.2 벡토르자료형의 구축자

- MPI_Type_vector
  벡토르자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_vector(  int count,   int blocklen,   int stride,
        MPI_Datatype old_type,   MPI_Datatype *newtype )
```

입구파라메터

count	블로크개수(부아닌 옹근수)
blocklen	매 블로크의 요소수(부아닌 옹근수)
stride	매 블로크의 시작과 시작사이거리(요소수)
oldtype	낡은 자료형(손잡이)

출구파라메터

newtype : 새 자료형(손잡이)

[실례 6.4] oldtype가 대역폭 16 을 가지고 형 {(double, 0), (char, 8)}로 넘기기된다고 하자. 이때 MPI_Type_vector(2, 3, 4, oldtype, newtype)를 호출하면 다음과 같이 넘기기한 자료형 newtype이 생성된다.

{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40), (double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104)}

때로는 우의 함수에서 파라메터 stride를 요소수대신에 구체적인 바이트수로 표현하면 더 편리할수 있다. 아래 함수에서는 stride를 바이트수로 준다.

- MPI_Type_hvector
  바이트변위의 벡토르자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_hvector( int count,   int blocklen, MPI_Aint
```

stride,   MPI_Datatype old_type,   MPI_Datatype *newtype)

입구파라메터

count	블로크개수(부아닌 옹근수)
blocklen	매 블로크에서의 요소수(부아닌 옹근수)
stride	매 블로크의 시작과 시작사이거리(바이트단위)
old_type	낡은 자료형(손잡이)

출구파라메터

newtype : 새 자료형(손잡이)

2.3 첨수화된 자료형구축자

- MPI_Type_indexed

첨수화된 자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_indexed(int count, int blocklens[], int indices[],
                     MPI_Datatype old_type, MPI_Datatype *newtype )
```

입구파라메터

count	블로크개수(옹근수), 첨수와 블로크길이배렬에서 항목의 수
blocklens	매 블로크의 요소수(부아닌 옹근수배렬)
indices	낡은 형의 단위로 측정한 매 블로크의 변위(옹근수배렬)
old_type	낡은 자료형(손잡이)

출구파라메터

newtype : 새 자료형(손잡이)

[실례 6.5] oldtype가 대역폭 16을 가지고 형{(double, 0), (char, 8)}로 넘기기된다고 하자. 그리고 B=(3, 1), D=(4, 0)이라고 하자. MPI_Type_indexed(2, B, D, oldtype, newtype)를 호출하면 다음과 같이 넘기기되는 자료형 newtype이 생성된다.

{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104), (double, 0), (char, 8)}

아래의 실례는 사용자정의자료형이 행렬의 웃삼각부분만을 전송하는데 어떻게 리용되는가를 보여준다.

[실례 6.6]

```
…
double a[100][100], disp[100], blocklen[100];
MPI_Datatype upper;
/* 행렬에서 매 행의 시작과 크기 계산(대각선으로부터 시작) */
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);
/* 전송 */
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
 …
```

- MPI_Type_hindexed

바이트변위의 첨수화된 자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_hindexed(int count, int blocklens[],  MPI_Aint indices[], MPI_Datatype old_type, MPI_Datatype *newtype )
```

입구파라메터

count	블로크수 - 첨수와 블로크길이배렬에서 항목의 수
blocklens	매 블로크의 요소수(부아닌 옹근수배렬)
indices	매 블로크의 바이트변위(MPI_Aint배렬)
old_type	낡은 자료형(손잡이)

출구파라메터

newtype : 새 자료형(손잡이)

[실례 6.7] oldtype가 대역폭 16을 가지고 형 {(double, 0), (char, 8)}로 넘기기된다고 하자. 그리고 B=(3, 1), D=(4, 0)이라고 하자. MPI_Type_hindexed(2, B, D, oldtype, newtype)를 호출하면 다음과 같이 넘겨지는 자료형 newtype이 생성된다.

{(double, 4), (char, 12), (double, 20), (char, 28), (double, 36), (char, 44), (double, 0), (char, 8)}

2.4 구조적인 자료형구축자

- MPI_Type_struct
  구조화된 자료형을 창조한다.

문법

```
#include "mpi.h"
int MPI_Type_struct( int count, int blocklens[], MPI_Aint
 indices[], MPI_Datatype old_types[], MPI_Datatype *newtype )
```

입구파라메터

count 블로크의 수(옹근수)

blocklens 매 블로크에서 요소의 수(배렬)

indices 매 블로크의 바이트변위(배렬)

old_types 매 블로크에서 요소의 형(자료형객체들에 대한 손잡이배렬)

출구파라메터

newtype : 새로운 자료형(손잡이)

[실례 6.8] type1 이 대역폭 16 을 가지고 형 {(double, 0), (char, 8)}로 넘기기된다고 하자. 그리고 B=(2, 1, 3), D=(0, 16, 26), T=(MPI_FLOAT, type1, MPI_CHAR)이라고 하자.
MPI_Type_struct(3, B, D, T, newtype)를 호출하면 다음과 같이 넘기기되는 자료형 newtype이 생성된다.

{(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char,

28)}

[실례 6.9] 구조행렬의 전송

```
struct Partstruct
{ char class;        /* 요소의 클라스 */
  double d[6];       /* 요소의 자리표 */
  char b[7];          /* 일련의 보충정보 */
};

struct Parstruct particle[1000];
int   i, desk, rank;
MPI_Comm comm;

/* 구조체서술자료형구축 */
MPI_Datatype Particletype;
MPI_Datatype type[3]={MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int blocklen[3]={1, 6, 7};
MPI_Aint disp[3]={0, sizeof(double), 7*sizeof(double)};

/* 자료형구축 */

MPI_Type_struct(3, blocklen, disp, type,   &Particletype);
MPI_Type_commit(&Particletype);
…
/* 행렬요소들을 전송 */
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
…
```

3. 사용자자료형의 리용

3.1 형넘기기와 해방

- MPI_Type_commit

사용자정의자료형의 사용시작을 선언한다.

문법

```
#include "mpi.h"
int MPI_Type_commit( MPI_Datatype *datatype )
```

입구파라메터

datatype : 자료형(손잡이)

- MPI_Type_free

사용자가 정의한 자료형을 해방한다.

문법

```
#include "mpi.h"
int MPI_Type_free( MPI_Datatype *datatype )
```

입구파라메터

datatype : 해방할 자료형(손잡이)

[실례 6.10.] MPI_Type_commit와 MPI_Type_free함수의 리용실례

```
int type1, type2;
MPI_Type_contuguous(5, MPI_FLOAT<&type1);
                                        /* 새형의 객체창조 */
MPI_Type_commit(&type1);
                   /* 새 type1 은 자료교환에 리용할수 있다. */

type2=type1;
                       /* type2 를 자료교환에 리용할수 있다. */
MPI_Type_vector(3, 5, 4, MPI_FLOAT, &type1);
                                           /* 새형의 객체창조 */
MPI_Type_commit(&type1);
                        /*새 type1 은 교환에 리용할수 있다. */
MPI_Type_free(&type2);                       /* 형의 해방 */
```

```
type2=type1;                /* type2는 자료교환에 리용가능 */
MPI_Type_free(&type2);
                         /*type1, type2 는 사용불가능; type2 는
                       MPI_DATATYPE_NULL, type1 은 미결정 */
```

- MPI_Get_count

"웃준위"요소의 수를 얻는다.

문법

```
#include "mpi.h"
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                                                   int *count )
```

입구파라메터

status   수신연산의 복귀상태(상태)

datatype 매 완충기요소의 자료형(손잡이)

출구파라메터

count : 수신된 요소의 수(옹근수)

3.2 통보길이를 주는 함수와 주소함수

- MPI_Get_elements

자료형에서 기초자료형요소의 수를 돌려준다.

문법

```
#include "mpi.h"
int MPI_Get_elements( MPI_Status *status, MPI_Datatype
                                   datatype,    int *elements )
```

입구파라메터

status   수신연산의 복귀상태(상태)

datatype 수신연산에 리용된 자료형(손잡이)

출구파라메터

count : 수신된 기초형의 요소수(옹근수)

[실례 6.11] MPI_Get_count와 MPI_Get_element의 리용

```
…
MPI_Type_contiguous(2, MPI_FLOAT, &type2);
MPI_Type_commit(&type2);
…
MPI_Comm_rank(comm, &rank);
if (rank == 0) {
    MPI_Send(a, 2, MPI_FLOAT, 1, 0, comm);
    MPI_Send(a, 3, MPI_FLOAT, 1, 1, comm);
}
else {
   MPI_Recv(a, 2, type2, 0, 0, comm, &start);
   MPI_Get_count(start, type2, &I);   /* i=1을 복귀 */
   MPI_Get_element(start, type2, &I); /* i=2를 복귀 */
   MPI_Recv(a, 2, type2, 0, 1, comm, &start);
   MPI_Get_count(start, type2, &I);
                              /* i=MPI_UNDEFINED을 복귀 */
   MPI_Get_element(start, type2, &I); /* i=3을 복귀 */
}
```

- MPI_Address

기억기에서 위치주소를 얻는다.

문법

```
#include "mpi.h"
int MPI_Address( void *location, MPI_Aint *address)
```

입구파라메터

location : 호출자(프로쎄스)기억기에서 위치

출구파라메터

address : 위치의 주소

[실례 6.12] 행렬에 MPI_Address함수를 리용, DIFF값은 909* sizeof(float)이고 i1, i2 값은 실행과 관련된다.

```
float a[100][100];
int i1, i2, DIFF;
MPI_Address(a, &i1);
MPI_Address(a[9][9], &i2);
DIFF=i2-i1;
```

[실례 6.13] 우의 실례에서 구조적의존성을 피하기 위한 코드변경
구조체요소들의 변위를 계산하기 위하여 MPI_Address함수를 호출한다.

```
struct Partstruct {
   char class;        /* 요소의 클라스 */
   double d[6];       /* 요소의 자리표 */
   char b[7];         /* 일련의 보충정보 */
};

struct Parstruct particle[1000];
int   i, desk, rank;
MPI_Comm comm;
MPI_Datatype type[3]={MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int blocklen[3]={1, 6, 7};
MPI_Aint disp[3];
/* 변위계산 */
MPI_Address(particle, &disp[0]);
MPI_Address(particle[0], &disp[1]);
MPI_Address(particle[1], &disp[2]);
for (i=2; i>=0; i--)
    disp[i]-=disp[0];
/* 자료형구축 */
MPI_Type_struct(3, blocklen, disp, type,  &particletype);
MPI_Type_commit(&particletype);
```

```
…
/* 행렬원소들을 전송 */
MPI_Send(particle, 1000, particletype, dest, tag, comm);
…
```

[실례 6.14] 우의 실례 7.8 에서 프로그람이 particletype의 대역을 정확히 설정하도록 다음과 같이 변경시킨다.

```
struct Partstruct {
   char class;         /* 요소의 클라스 */
   double d[6];       /* 요소의 자리표 */
   char b[7];          /* 일련의 보충정보 */
};

struct Parstruct particle[1000];
int   i, desk, rank;
MPI_Comm comm;
MPI_Datatype particletype;
MPI_Datatype type[3]={MPI_CHAR,
                              MPI_DOUBLE, MPI_CHAR, MPI_UB};
int blocklen[4]={1, 6, 7, 1};
MPI_Aint disp[4];
/* 구조요소들의 변위계산 */
MPI_Address(particle, &disp[0]);
MPI_Address(particle[0].d, &disp[1]);
MPI_Address(particle[0].b, &disp[2]);
MPI_Address(particle[1].b, &disp[3]);

for (i=3; i>=0; i--)
    disp[i]-=disp[0];
/* 자료형구축 */
MPI_Type_struct(4, blocklen, disp, type, &particletype);
MPI_Type_commit(&particletype);
…
```

```
/* 행렬요소들을 전송 */
MPI_Send(particle, 1000, particletype, dest, tag, comm);
…
```

4. 자료의 묶기와 풀기

- MPI_Pack
  련속적인 기억구역으로 자료형을 묶는다.

문법

```
#include "mpi.h"
int MPI_Pack(void *inbuf, int incount,   MPI_Datatype datatype,
    void *outbuf, int outcount, int *position, MPI_Comm comm )
```

입구파라메터

inbuf 입구완충기시작(선택)
incount 입구자료항목개수(옹근수)
datatype 매 입구자료항목의 자료형(손잡이)
outcount 바이트단위의 출구완충기크기(옹근수)
position 완충기에서의 현재 위치(옹근수형지적자)
comm 묶어진 통보를 위한 통신기(손잡이)

출구파라메터

outbuf : 출구완충기의 시작주소(선택)

- MPI_Pack_size
  통보를 묶는데 필요한 공간크기의 웃한계값을 준다.

문법

```
#include "mpi.h"
int MPI_Pack_size( int incount, MPI_Datatype datatype,
                                    MPI_Comm comm,   int *size )
```

입구파라메터

incount   묶기호출에서의 파라메터개수(옹근수)
datatype 묶기호출에서의 파라메터들의 형(손잡이)
comm   묶기호출에서의 통신기(손잡이)

출구파라메터
size : 묶어진 통보크기의 웃한계(바이트단위)(옹근수)

- MPI_Unpack

편속기억구역에로 자료형을 풀기한다.

문법

```
#include "mpi.h"
int MPI_Unpack( void *inbuf, int insize, int *position,
            void *outbuf, int outcount, MPI_Datatype datatype,
                                            MPI_Comm comm )
```

입구파라메터
inbuf   입구완충기시작(선택)
insize   입구완충기의 바이트크기(옹근수)
position 현재 위치(옹근수형지적자)
outcount 풀어야 할 항목개수(옹근수)
datatype 매 출구자료항목의 자료형(손잡이)
comm   묶은 통보에 대한 통신기(손잡이)

출구파라메터
outbuf : 출구완충기의 시작주소

[실례 6.15] 다음의 두 프로그람은 동일한 통보를 발생한다.

```
int i;
char c[100];
int disp[2];
int blocklen[2]={1, 100};
MPI_Datatype type[2]={MPI_INT, MPI_CHAR};
```

```
MPI_Datatype Type;
/* 자료형구축 */
MPI_Address(&i, &disp[0]);
MPI_Address(c, &disp[1]);
MPI_Type_struct(2, blocklen, disp, type, &Type);
MPI_Type_commit(&Type);
/* 전송 */
MPI_Send(MPI_BOTTOM, 1, Type, 1, 0, MPI_COMM_WORLD);
```

자료묶기를 진행한다.

```
int i;
char c[100];
char buffer[110];
int position = 0;
/* 묶기 */
MPI_Pack(&i, 1, MPI_INT, buffer, 110,
                                    &position, MPI_COMM_WORLD);
MPI_Pack(c, 100, MPI_CHAR, buffer, 110,
                                    &position, MPI_COMM_WORLD);
/* 전송 */
MPI_Send(buffer, position, MPI_PACKED,
                                            1, 0, MPI_COMM_WORLD);
```

[실례 6.16] 우의 실례에서 보낸 자료를 받는(푸는) 프로그람

```
int i;
char c[100];
MPI_Status status;
int disp[2];
int blocklen[2]={1, 100};
MPI_Datatype type[2]={MPI_INT, MPI_CHAR};
MPI_Datatype Type;
/* 자료형구축 */
MPI_Address(&i, &disp[0]);
```

```
MPI_Address(c, &disp[1]);
MPI_Type_struct(2, blocklen, disp, type, &Type);
MPI_Type_commit(&Type);
/* 수신 */
MPI_Recv(MPI_BOTTOM, 1, Type, 0,
                                        0, MPI_COMM_WORLD, &status);
```

자료풀기를 진행한다.

```
int i;
char c[100];
MPI_Status status;
char buffer[110];
int position = 0;
/* 수신 */
MPI_Recv(buffer, 110, MPI_PACKED, 1, 0,
                                       MPI_COMM_WORLD, &status);
/* 풀기 */
MPI_Unpack(buffer, 110, &position, &i, 1, MPI_INT, MPI_COMM_
WORLD);
MPI_Unpack(buffer, 110, &position, c, 100, MPI_CHAR, MPI_COMM_
WORLD);
```

제 7 절 환경조종

1. 실행과 관련한 정보

MPI실행환경을 서술하는 속성들은 MPI를 초기화한 후에 통신기 MPI_COMM_WORLD에 련결된다. 이 속성들을 제거하거나 값들을 변경하는것은 오유이다.

미리 정의되여있는 기본속성들은 다음과 같다.

MPI_TAG_UB - 표적값의 웃한계

MPI_HOST - 주프로쎄스(host)의 번호, 만일 없으면 MPI_PROC_NULL로 된다.
MPI_IO - 표준입출구마디의 번호(호출프로쎄스의 번호도 가능)
같은 통신기의 마디들은 이 파라메터값들을 각이하게 줄수 있다.
MPI_WTIME_ISGLOBAL - 시계들이 동기화되였는가를 지적하는 론리변수

이 미리 정의된 속성들은 MPI실행환경초기화(MPI_Init) 및 MPI실행환경끝내기(MPI_Finalize)의 값들을 변화시키지 않는다.
먼저 처리기이름을 얻는 함수에 대하여 보기로 한다.

- MPI_Get_processor_name
처리기의 이름을 얻는다.

문법
```
#include "mpi.h"
int MPI_Get_processor_name(char *name, int *resultlen)
```

출구파라메터
name 현존 물리적마디점에 대한 단일식별자
적어도 MPI_MAX_PROCESSOR_NAME 크기의 배렬이여야 한다.
resultlen 이름(문자렬)의 길이

2. 계수기와 동기화

MPI는 계수기를 결정한다.
병렬프로그람에서 시간선택이 오유수정 등에서 매우 중요하고 또한 현존 계수기들(POSIX에서 1003.1-1988, 1003.4D 14.1 등)이 쓰기 불편하고 고가용성계수기들에 상응한 호출을 하지 못함으로 계수기가 리용된다.

- MPI_Wtime
호출한 처리기에서 경과된 시간을 돌려준다.

문법

```
#include "mpi.h"
double MPI_Wtime()
```

귀환값

이전의 어떤 시각으로부터 경과된 시간을 초단위로 돌려준다. “이전 시간”은 프로쎄스가 살아있는 동안은 변하지 않는다.

- MPI_Wtick

MPI_Wtime의 분해능을 돌려준다.

문법

```
#include "mpi.h"
double MPI_Wtick()
```

귀환값

경과한 시간을 초단위로 하는 분해능

3. 초기화와 완료

MPI에서 기본목적의 하나는 MPI원천코드들의 이식가능성을 보장하는데 있다. 이것은 프로그람이 MPI를 리용하고 표준언어코드를 실행하면서 작성한 그 형태로 이식가능하여야 하며 다른 체계로 이식할 때 원천코드를 조금도 변경시키지 말아야 한다는것을 의미한다. 이것은 MPI프로그람을 지령행에 의하여 시작하거나 끝내지 말아야 하며 MPI프로그람이 실행되는 환경을 사용자가 변경시키지 않도록 해야 한다는것이다. 대신 다른 MPI함수들을 호출하기전에 일련의 설정을 하여야 한다. 이 함수가 MPI_Init함수이며 모든 MPI프로그람은 이 함수호출로부터 시작된다. MPI_Finalize함수는 모든 MPI설정을 해제한다.

- MPI_Init

MPI실행환경을 초기화한다.

문법

```
#include "mpi.h"
```

int MPI_Init(int *argc, char ***argv)

입구파라메터

argc 파라메터개수 지적자

argv 파라메터벡토르지적자

- MPI_Finalize

MPI실행환경을 끝낸다.

문법

```
#include "mpi.h"
int MPI_Finalize()
```

설명

모든 MPI프로세스는 끝나기전에 이 루틴을 반드시 호출해야 한다. 이 함수가 호출된 후에는 어떤 MPI함수도 호출하지 말아야 한다. 따라서 MPI_Finalize를 호출한 후에는 return을 호출하는것외에 더 다른것을 하지 말아야 한다.

- MPI_Initialized

MPI_Init가 호출되였는가 아닌가를 검사한다.

문법

```
#include "mpi.h"
int MPI_Initialized( int *flag )
```

출구파라메터

flag : MPI_Init가 호출되였으면 참 아니면 거짓이다.

- MPI_Abort

MPI프로세스들을 끝낸다.

문법

```
#include "mpi.h"
```

```
int MPI_Abort( MPI_Comm comm, int errorcode)
```

입구파라메터

comm : 중지시키려는 과제들의 통신기
errorcode : 현재의 가동환경에 반환하는 오유코드

설명

대부분 체계에서 통신기 comm에 속한 모든 MPI프로쎄스들을 끝낸다.

- MPI_Get_version

MPI의 판본을 얻는다.

문법

```
#include "mpi.h"
int MPI_Get_version( int *version,   int *subversion )
```

출구파라메터

version MPI의 기본판본(1 또는 2)
Subversion MPI의 부분판본

설명

정의된 값 MPI_VERSION과 MPI_SUBVERSION들은 같은 정보를 포함한다. 이 루틴들은 mpi.h와 mpif.h에서 서고가 판본과 일치하는가를 검사한다.

4. 오유처리함수들

MPI는 사용자에게 안전한 통보전송을 제공한다. 그러나 만일 MPI 응용프로그람실행환경이 비정상적으로 구축되여있다면 MPI병렬계산환경 구축자들은 가능한 오유들을 처리하여야 한다. MPI는 MPI호출시에 만나게 되는 일련의 오유들을 처리할수 있다.

4.1 오유처리기

사용자는 오유처리기를 통신기와 련결할수 있다. MPI에는 미리 정의된 다음의 몇가지 오유처리기가 있다.

- MPI_ERRORS_ARE_FATAL : 이 오유처리기는 실행중의 모든 프로쎄스들을 중지시킨다. 이것은 MPI_Abort함수를 통신기파라메터를 MPI_COMM_WORLD로 지정하여 호출한것과 같다.
- MPI_ERRORS_RETURN : 사용자에게 오유코드를 주는것외에 그 어떤 작용도 없다.

MPI는 새 오유처리기를 창조하고 그것을 통신기와 련결하거나 삭제하는 함수들을 제공한다.

- MPI_Errhandler_create

MPI형 오유처리기를 창조한다.

문법

```
#include "mpi.h"
int MPI_Errhandler_create(MPI_Handler_function *function,
                          MPI_Errhandler    *errhandler)
```

입구파라메터

function : 사용자정의오유처리함수

출구파라메터

errhandler : MPI오유처리기(손잡이)

- MPI_Errhandler_free

MPI형오유처리기를 해방한다.

문법

```
#include "mpi.h"
int MPI_Errhandler_free( MPI_Errhandler *errhandler )
```

입구파라메터

errhandler : MPI오유처리기(손잡이)

탈퇴시 MPI_ERRHANDLER_NULL로 된다.

- MPI_Errhandler_get

통신기에 련결된 오유처리기를 얻는다.

문법

```
#include "mpi.h"
int MPI_Errhandler_get( MPI_Comm comm, MPI_Errhandler
                                        *errhandler )
```

입구파라메터

comm : 오유처리기를 얻으려는 통신기(손잡이)

출구파라메터

errhandler : 현존 통신기와 련결된 MPI오유처리기(손잡이)

- MPI_Errhandler_set

오유처리기를 통신기에 설정한다.

문법

```
#include "mpi.h"
int MPI_Errhandler_set( MPI_Comm comm, MPI_Errhandler
                                        errhandler )
```

입구파라메터

comm	오유처리기를 설정하여야 할 통신기(손잡이)
errhandler	새로운 MPI 오유처리기(손잡이)

4.2 오유코드들

모든 MPI함수들은 성공적으로 실행(MPI_SUCCESS) 혹은 MPI오유클라스와 관련한 정보를 얻기 위한 코드를 귀환한다. MPI함수가 몇가지 각이한 연산들에 의하여 끝나 몇개의 독립적인 오유가 발생하면 MPI 함수는 다중오유코드를 준다.

- MPI_Error_string
  주어진 오유코드에 대한 문자렬을 복귀한다.

문법
  #include "mpi.h"
  int MPI_Error_string(int errorcode, char *string, int
                        *resultlen)

입구파라메터
  errorcode : MPI함수 또는 MPI오유클라스에 의해 귀환되는 오유코드

출구파라메터
  string   오유코드에 대응되는 본문
  resultlen 문자렬의 길이

- MPI_Error_class
  오유코드를 오유클라스로 변환한다. 매 오유코드를 표준오유코드(오유클라스)로 넘기고 다음에는 매개 표준오유코드를 자기에게로 넘긴다.

문법
  #include "mpi.h"
  int MPI_Error_class(int errorcode, int *errorclass)

입구파라메터
  errorcode : MPI함수에 의해 복귀된 오유코드

출구파라메터
  errorclass : 오유코드와 관련되는 오유클라스

△ MPI오유클라스는 다음과 같다.

MPI_SUCCESS	오유없음
MPI_ERR_BUFFER	완충기지적자오유

MPI_ERR_COUNT	개수파라메터오유
MPI_ERR_TYPE	자료형파라메터오유
MPI_ERR_TAG	표적파라메터오유
MPI_ERR_COMM	통신기오유
MPI_ERR_RANK	프로쎄스번호오유
MPI_ERR_REQUEST	틀린 요구
MPI_ERR_ROOT	뿌리오유
MPI_ERR_GROUP	그룹오유
MPI_ERR_OP	연산자오유
MPI_ERR_TOPOLOGY	위상오유
MPI_ERR_DIMS	데카르트구조파라메터오유
MPI_ERR_ARG	자료형불일치오유
MPI_ERR_UNKNOWN	알수 없는 오유
MPI_ERR_TRUNCATE	수신시 통보가 잘리우는 오유
MPI_ERR_OTHER	목록에 없는 오유
MPI_ERR_INTERN	MPI내부오유
MPI_ERR_IN_STATUS	상태오유코드
MPI_ERR_PENDING	요구미결
MPI_ERR_LASTCODE	마지막오유코드

제 3 장 MPI 프로그람작성실기

제 1 절 기초위상 MPI_COMM_WORLD 에서의 프로그람작성

이 절의 목적은 완전그라프인 기초위상 MPI_COMM_WORLD에서 병렬프로쎄스들의 호상작용수법을 습득하는데 있다.

[실례 1] hello.c

- 프로그람의 작성

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
 {
     MPI_Init(&argc, &argv);
     printf("Hello, world\n");
     MPI_Finalize();
     return 0;
 }
```

- 프로그람을 콤파일

```
~>mpicc -o hello hello.c
```

만일 오유가 있으면 편집기에서 프로그람을 수정한다.

- 프로그람의 실행

```
~> mpirun -np 4 hello
```

화면에는 다음과 같은 결과가 현시된다.

```
Hello, world
Hello, world
Hello, world
Hello, world
```

지령행에서 4 라는것은 4 개의 독립적인 프로쎄스에서 프로그람을 실행한다는것을 의미한다. 즉 프로그람은 4 개의 가지로 되여있다. 앞으로 "병렬프로쎄스"와 "병렬프로그람의 가지"를 같은 말로 취급한다.

mpirun의 의미를 보면 다음과 같다. 프로그람 hello는 이미 구축된 4 개의 가상콤퓨터들에 실행전에 복사되며 매 복사된 프로그람을 실행한다.

[실례 2] 병렬프로그람의 매개 가지는 병렬프로쎄스의 식별번호와 크기 즉 병렬프로그람의 매개 가지가 들어있는 가상콤퓨터의 수를 화면에 출력한다.

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
 {
    int size, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("size=%d, rank=%d\n", size, rank);
    MPI_Finalize();
    return 0;
 }
```

프로그람을 콤파일하고 실행시키는 과정은 우의 실례에서와 같다.

[실례 3] 0 번 프로세스는 7 번 프로세스에 통보(이 경우에는 식별자)를 보낸다. 7 번 프로세스는 자기의 식별자와 0 번 프로세스로부터 받은 식별자를 화면에 출력한다.

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
 {
     int size, rank, r;
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     if (rank == 0)
      MPI_Send(&rank, 1, MPI_INT, 7, 2, MPI_COMM_WORLD);
     else
      if (rank == 7) {
            MPI_Recv(&r, 1, MPI_INT, 0, 2, MPI_COMM_WORLD,
&status);
            printf("process=%d r=%d\n", rank, r);
      }

   MPI_Finalize();
   return 0;
}
```

[실례 4] 0 번 프로세스는 1 번 프로세스에 통보(자기의 식별자)를 보내고 1 번 프로세스는 2 번 프로세스에 받은 번호를 보내며 2 번 프로세스는 받은 번호를 3 번 프로세스에로 보내는 등 사슬을 따라 번호증가방향으로

계속한다. 마지막으로 0 번 프로세스는 size-1 번 프로세스로부터 번호를 받아 화면에 자기의 번호와 수신된 정보를 출력한다. 즉 정보를 콤퓨터 "고리"를 따라 전달한다.

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
  {
   int size, rank, r;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
        MPI_Send(&rank, 1, MPI_INT, rank+1, 2, MPI_COMM_WORLD);
        MPI_Recv(&r, 1, MPI_INT, size-1, 2, MPI_COMM_WORLD,
&status);
        printf("process= %d r=%d\n", rank, r);
   }
   else{
        MPI_Recv(&r, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD,
&status);
        MPI_Send(&r, 1, MPI_INT, (rank+1)%size, 2,
MPI_COMM_WORLD);
   }
   MPI_Finalize();
   return 0;
}
```

제 2 절 각이한 위상들에서의 프로그람작성

목적 : 데카르트위상과 “그라프”위상에서 실행되는 MPI프로그람들을 작성하는데 있다.

[실례 5] 병렬체계의 모든 가지들은 콤퓨터자리표증가방향으로 린접가지들에 자료를 전송한다.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define DIMS 2

main(int argc, char **argv)
 {
    int size, rank, i, A, B, dims[DIMS];
    int periods[DIMS], sourc1, sourc2, dest1, dest2;
    int reorder = 0;
    MPI_Comm comm_cart;
    MPI_Status status;

   MPI_Init(&argc, &argv);
/* 매 프로쎄스는 프로쎄스의 총개수를 알아낸다. */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* 그리고 자기의 번호를 알아낸다. */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    A=rank;
    B=-1;
/* dims배렬을 초기화하고 배렬 periods을 “2 차원” 위상으로 채운다. */
     for (i=0; i<DIMS; i++) {
        dims[i]=0;
        periods[i]=1;
     }
```

```
/* 살창의 크기를 지적하는 dims배렬을 채운다. */
      MPI_Dims_create(size, DIMS, dims);
/* 통신기 comm_cart를 가진 "2 차원" 위상을 생성한다. */
      MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder, &comm_cart);
/* 매 프로쎄스는 자리표의 큰값방향으로 자리표를 따라 린접가지들을 찾는다. */
      MPI_Cart_shift(comm_cart, 0, 1, &sourc1, &dest1);
      MPI_Cart_shift(comm_cart, 1, 1, &sourc2, &dest2);
/* 매 프로쎄스는 자기의 자료(변수 A의 값)를 자리표의 큰 값을 취하는 린접프로쎄스들에 전송하고 "고리"를 따라 자리표의 작은 값을 취하는 린접가지로부터 자료를 받아 B에 넣는다. */
/* 매 프로쎄스는 자기의 번호(이 번호를 린접프로쎄스에 전송)와 변수 B의 값(린접프로쎄스의 번호)을 출력한다. */
      MPI_Sendrecv(&A, 1, MPI_INT, dest1, 2, &B, 1, MPI_INT, sourc1, 2, comm_cart, &status);
      printf("rank= %d B=%d\n", rank, B);
      MPI_Sendrecv(&A, 1, MPI_INT, dest2, 2, &B, 1, MPI_INT, sourc2, 2, comm_cart, &status);
      printf("rank= %d B=%d\n", rank, B);
/* 모든 프로쎄스들은 위상 comm_cart와 련관된 모든 프로쎄스들을 완료하고 프로그람을 끝낸다. */
      MPI_Comm_free(&comm_cart);
      MPI_Finalize();
      return 0;
}
```

[실례 6] 실례 4 와 비슷하지만 위상을 리용한다. 0 번 프로쎄스는 콤퓨터 "고리"를 따라 자료전송을 시작하며 전송되는 자료들은 모든 콤퓨터들을 따라 순차적으로 "통과하여" 0번 프로쎄스를 실행하는 0번 콤퓨터로 돌아온다.

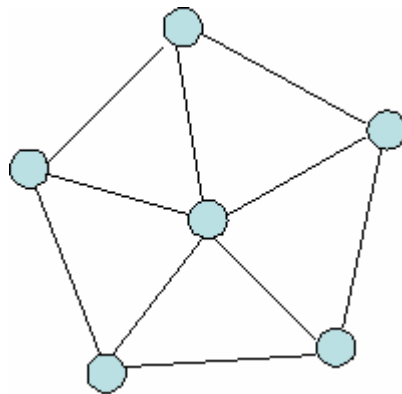
```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "mpi.h"

#define DIMS 1

main(int argc, char **argv)
  {
     int size, rank, i, A, B, dims[DIMS];
     int periods[DIMS], source, dest;
     int reorder = 0;
     MPI_Comm comm_cart;
     MPI_Status status;

     MPI_Init(&argc, &argv);
/* 매 프로쎄스는 프로쎄스의 총개수를 알아낸다. */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
/* dims배렬을 초기화하고 배렬 periods을 “2 차원” 위상으로 채운다. */
    for (i=0;i < DIMS; i++) {
      dims[i]=0;
      periods[i]=1;
    }
 /* 살창의 크기를 지적하는 dims배렬을 채운다. */
    MPI_Dims_create(size, DIMS, dims);
/* 통신기 comm_cart를 가진  “고리”형위상을 생성한다. */
    MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder,
&comm_cart);
/* 자기의 번호를 알아낸다. */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    A=rank;
    B=-1;
/* 매 프로쎄스는 고리를 따라 번호가 큰방향으로 자기의 이웃들을 찾는다.
*/
    MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
/* 0 번 프로쎄스는 고리를 따라 자료(자기번호)를 전송하며 size-1 번
프로쎄스로부터 이 값을 받는다. */
```

```
    if (rank == 0) {
        MPI_Send(&A, 1, MPI_INT, dest, 12, comm_cart);
        MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
        printf("rank=%d B=%d\n", rank, B);
    }
/* 기타 모든 나머지 프로세스들에서의 동작 */
    else {
        MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
        MPI_Send(&B, 1, MPI_INT, dest, 12, comm_cart);
    }
/* 완료작업 */
    MPI_Comm_free(&comm_cart);
    MPI_Finalize();
    return 0;
}
```

[실례 7] N개의 콤퓨터들로 "별"형위상을 구성한다. 여기서 0 번콤퓨터는 뿌리프로쎄스이고 "잎"들은 아래 그림과 같이 련결되며 고리를 이룬다. 뿌리프로쎄스는 "잎"들에 자료를 보내고 또 그것들은 수신된 자료를 처리하여 뿌리프로쎄스로 보낸다. 뿌리프로쎄스는 일련의 정보를 출력한다.

"잎"들에서는 임의의 위상을 구성할수 있는데 이 실례의 경우에는 "고리"위상이다.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "mpi.h"

#define DIMS 1

main(int argc, char **argv)
  {
    int size, size1, rankgr, i, v, j, key, color, A, *B, C;
    int periods[DIMS], dims[DIMS], *index, *edges, source, dest, D[2];
    int reorder = 0;
    MPI_Comm comm_cart, comm_gr, comm_0, comm_1;
    MPI_Status st;

    MPI_Init(&argc, &argv);
/* 매 프로쎄스는 프로쎄스의 총개수를 알아낸다. */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* 그라프위상에서 정점과 호를 서술하기 위한 배렬기억할당 */
    index=(int *)malloc(size*sizeof(int));
    edges=(int *)malloc((size-1)*(size-1)*3*sizeof(int));
/* 그라프위상의 정점과 호를 서술하기 위한 배렬의 값을
넣고 “그라프”위상을 제시한다. */
    index[0]=size-1;
    for (i=0; i<size-1; i++) edges[i]=i+1;
    v=0;
for (i=1;i<size;i++) {
        index[i]=(size-1)+i*3;
        edges[(size-1)+v++]=0;
        edges[(size-1)+v++]=((i-2)+(size-1))%(size-1)+1;
        edges[(size-1)+v++]=i%(size-1)+1;
}
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reorder,
&comm_gr);
/* 매 프로쎄스는 자기의 번호를 식별한다. */
    MPI_Comm_rank(comm_gr, &rankgr);
    if (rankgr == 0) {
```

```
      color=MPI_UNDEFINED;
      key=0;
      MPI_Comm_split(comm_gr, color, key, &comm_0);
    }
    else {
     color=1;
     key=rankgr;
     MPI_Comm_split(comm_gr, color, key, &comm_1);
    }
/* 매 프로쎄스는comm_1 에서의 자기번호를 식별한다. */
    MPI_Comm_size(comm_1, &size1);
/* comm_1 에서 이제는 "고리"형위상 comm_cart를 생성한다. */
/* 배렬 dims를 초기화하고"고리"형위상을 위한 배렬 periods에
    값을 넣는다. */
    for (i=0; i < DIMS; i++) {
        dims[i]=0;
        periods[i]=0;
    }
/*  (1 차원)살창의 크기를 지정하는 배렬 dims에 값을 넣기 */
    MPI_Dims_create(size1, DIMS, dims);
/* "고리"comm_cart를 가진 "고리"형위상을 생성한다. */
    MPI_Cart_create(comm_1, DIMS, dims, periods, reorder,
&comm_cart);
/* 매 프로쎄스는 고리를 따라 번호가 커지는 방향으로 이웃프로쎄스를
   찾는다. */
   MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
/* 0 번 프로쎄스는 자기들끼리 "고리"를 형성하는"잎"들에
   자료(자기의 번호값)를 전송한다. */
   D[0]=0; D[1]=0;
   C=0; A=0; B=(int *)malloc(2 * size * sizeof(int));
   for (i=0; i<2; i++) {
     B[i]=0;
   /* 자료를 "잎"들에 보내기 */
        MPI_Bcast(&A, 1, MPI_INT, 0, comm_gr);
```

```
   /* “잎” 들로부터의 자료수집 */
        MPI_Gather(D, 2, MPI_INT, B, 2, MPI_INT, 0, comm_gr);
        printf("B=%d\n", B[i]);
    }
/* 나머지 프로쎄스들에서의 동작 */
  MPI_Bcast(&A, 1, MPI_INT, 0, comm_gr);
/*자료처리 */
  A=A+rankgr; D[0]=A;
  MPI_Sendrecv(&A, 1, MPI_INT, dest,12, &C, 1, MPI_INT, source, 12,
comm_1, &st);
  D[1]=C;
 /* 뿌리등록부에 자료전송 */
  MPI_Gather(D, 2, MPI_INT, B, 2, MPI_INT, 0, comm_gr);
/* “잎” 들에서 위상을 해방*/
  MPI_Comm_free(&comm_cart);
  MPI_Comm_free(&comm_1);
/* 모든 프로쎄스들을 해방 */
  MPI_Comm_free(&comm_gr);
  MPI_Finalize();
  return 0;
}
```

제 3 절 행렬에 벡토르곱하기와 행렬에 행렬곱하기

목적: 행렬에 벡토르곱하기와 행렬에 행렬곱하기와 같은 알고리듬의 병렬화수법들을 습득하는데 있다.

[실례 8] “고리” 형위상에서 행렬에 벡토르를 곱하는 문제이다.

n1*n2 행렬 A와 n2차 벡토르 B가 주어졌을 때 C=A*B를 계산한다. 행렬 A와 벡토르 B를 일정한 크기로 자르고 매 프로쎄스는 자기의 부분을

생성한다. P1 개의 콤퓨터들로 이루어진 체계에서 프로그람을 실행하므로 행렬 A와 벡토르 B를 p1 개의 가로대역으로 나눈다. 행렬 A와 벡토르 B는 p1 로 완전히 나누인다고 가정한다.

프로그람에서는 p1=4 로 하였다.

```
#include     <stdio.h>
#include     <stdlib.h>
#include     <time.h>
#include     <sys/time.h>
#include     "mpi.h"
/*매 프로쎄스에 A와 B의 대역크기를 준다.*/
/* 이 크기가 모든 프로쎄스들에서 같다고 가정한다.*/
#define M 16
#define N 4
/* DIMS -데카르트위상의 크기, "고리"는 1 차원이다.*/
#define DIMS 1
#define EL(x) (sizeof(x)/sizeof(x[0]))
/* 대역변수 */
static double A[N][M], B[N],C[N];

main(int argc, char **argv)
 {
    int size, rank, i, j, k, i1, sour, dest;
    int dims[DIMS];
    int periods[DIMS];
    int reorder = 0;
    MPI_Comm comm_cart;
    MPI_Status st;
    double rt, t1, t2;

    MPI_Init(&argc, &argv);
/* 매 프로쎄스는 문제의 크기를 알아낸다. */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/*행렬 dims 초기화, "고리"형위상을 위한 행렬 periods값넣기*/
```

```
    for (i=0; i<DIMS; i++) {
      dims[i]=0;
      periods[i]=1;
    }
/* (1 차원)살창크기를 지적하는 dims행렬값넣기*/
     MPI_Dims_create(size, DIMS, dims);
/* 통신기 comm_cart를 가진 “고리”위상 생성*/
     MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods,
                                   reorder, &comm_cart);
/* 매 프로쎄스는 자기의 번호를 인식한다.*/
     MPI_Comm_rank(comm_cart, &rank);
/* 매 프로쎄스는 고리를 따라 번호가 작은 방향으로 자기의 이웃들을
찾는다.*/
     MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest);
  for (j=0; j<N; j++)
    for (i=0; i<M; i++) {
      A[j][i]=3.0;
      B[j]=2.0;
      C[j]=0.0;
    }
/* 시작시간설정 */
  t1=MPI_Wtime();
/* 매 프로쎄스는 자기 대역에서 곱하기를 진행 */
/* 맨 바깥순환 for (k) 는 콤퓨터순환 */
  for (k=0 ;k < size; k++) {
   int d;
   d=((rank+k)%size)*N;
   /* 매 프로쎄스는 자기 대역에서의 곱하기를 진행*/
      for (j=0; j<N; j++)
         for (i1=0, i=d; i<d+N;i++, i1++)
           C[j]+=A[j][i]*B[i1];
   }
   /* 매 프로쎄스는 낮은 번호의 린접프로쎄스에 B의 대역을 전송*/
   /* 즉 B의 대역을 고리를 따라 밀기한다. */
```

```
    MPI_Sendrecv_replace(B, EL(B), MPI_DOUBLE, dest, 12, sour, 12,
comm_cart, &st);
/* 곱하기 끝시간 측정*/
 t2=MPI_Wtime();
 rt=t2-t1;
    printf("rank=%d Time=%d\n", rank, rt);
/* 결과출력 */
 for (i=0;i<N;i++)
    printf("rank=%d RM=%6.2f\n", rank, C[i]);
 MPI_Comm_free(&comm_cart);
 MPI_Finalize();
  return 0;
}
```

[실례 9] "완전그라프" 위상에서 행렬에 벡토르를 곱하는 문제이다.

n1*n2 행렬 A와 n2차 벡토르 B가 주어졌을 때 C=A*B를 계산한다. 행렬 A와 벡토르 B를 일정한 크기로 자르고 매 프로쎄스는 자기의 부분을 생성한다. 매 콤퓨터에서 결과를 얻은 후 단일벡토르 B로 집합하여 모든 콤퓨터들에 전송한다(이것은 반복알고리듬에서 한가지 반복모형이다.). 프로그람에서는 콤퓨터대수를 4로 하였다.

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <time.h>
#include   <sys/time.h>
#include   "mpi.h"
/* 매 프로쎄스에 A와 B의 대역크기를 준다.*/
/* 이 크기가 모든 프로쎄스들에서 같다고 가정한다.*/

#define M 16
#define N 4
/* DIMS 4 데카르트위상의 크기, "고리"는 1차원이다.*/
#define DIMS 1
#define EL(x) (sizeof(x)/sizeof(x[0][0]))
```

```
static double A[N][M], B[N], C[N];

main(int argc, char **argv)
  {
    int size, rank, i, j;
    double rt, t1, t2;
     int sour, dest;
     int dims[DIMS];
     int periods[DIMS];
     int reorder = 0;

    MPI_Comm comm_cart;

    MPI_Init(&argc, &argv);
/* 매 프로세스는 문제의 크기를 알아낸다. */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* 매 프로세스는 자기의 번호를 알아낸다. */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* 매 프로세스는 A와 B의 대역을 생성, C의 대역을 초기화 */
    for (i=0;i<M;i++)
     for (j=0;j<N;j++) {
        A[j][i]=3.0;
        B[j]=2.0;
        C[j]=0.0;
    }
/*행렬 dims초기화, “고리”형위상을 위한 행렬 periods값넣기*/
     for (i=0; i<DIMS; i++) {
        dims[i]=0;
        periods[i]=1;
     }
 /* (1 차원)살창크기를 지적하는 dims행렬값넣기*/
       MPI_Dims_create(size, DIMS, dims);
 /* 통신기 comm_cart를 가진 “고리”위상 생성*/
```

```
    MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods,
                                reorder, &comm_cart);
/* 매 프로세스는 자기의 번호를 인식한다.*/
    MPI_Comm_rank(comm_cart, &rank);

/* 매 프로세스는 고리를 따라 번호가 작은 방향으로 자기의 이웃들을
찾는다. */
  MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest);
/* 시작시간설정 */
  t1=MPI_Wtime();
  for (j=0;j<N;j++)
   for (i=0; i<M; i++)  C[j]+=A[j][i]*B[j];

  MPI_Allgather(C, EL(C), MPI_DOUBLE, B, N, MPI_DOUBLE,
MPI_COMM_WORLD);
/*매 프로세스는 풀이시간을 출력, 0 번 프로세스는 벡토르 B를 출력*/
  t2=MPI_Wtime();
  rt=t2-t1;
  printf("rank=%d Time=%f\n", rank, rt);
/*결과출력*/
  if (rank == 0) {
   for (i=0; i< N; i++)
    printf("B = %6.2f\n", B[i]);
  }
  MPI_Finalize();
  return 0;
}
```

[실례 10] "고리"형위상에서 행렬에 행렬곱하기를 진행한다.

n1*n2 행렬 A와 n2*n3 행렬 B를 곱하여 n1*n3 행렬 C=A*B를 계산한다. 행렬 A, B를 미리 대역으로 가르고 매 프로세스는 자기의 부분을 계산한다.

p1 개의 콤퓨터들로 이루어진 "고리"에서 프로그람을 실현한다. 행렬 A를 p1 개의 가로대역으로 행렬 B를 p1 개의 세로대역으로 나눈다.

여기에서는 매 콤퓨터에 행렬 A의 하나의 대역과 행렬 B의 하나의 대역만을 넣는다고 가정한다. 그리고 행렬 A와 벡토르 B는 p1 로 완전히 나누인다고 가정한다.

프로그람에서는 p1=8 로 하였다.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include "mpi.h"

/*매 프로쎄스에 A와 B의 대역크기를 준다.*/
/* 이 크기가 모든 프로쎄스들에서 갈다고 가정한다.*/
#define M 320
#define N 40
/* DIMS 1 데카르트위상의 크기, "고리"는 1 차원이다.*/
#define DIMS 1
#define EL(x) (sizeof(x)/sizeof(x[0][0]))

static double A[N][M], B[M][N], C[N][M];

main(int argc, char **argv)
 {
    int size, rank, i, j, k, i1, j1, sour, dest;
    int dims[DIMS], periods[DIMS];
    int reorder = 0;
    MPI_Comm comm_cart;
    MPI_Status st;
    double rt, t1, t2;
    MPI_Init(&argc, &argv);
/* 매 프로쎄스는 문제의 크기를 알아낸다. */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* 행렬 dims초기화, "고리" 형위상을 위한 행렬 periods값 넣기*/
    for (i=0; i< DIMS; i++) {
```

```
        dims[i]=0;
          periods[i]=1;
     }
/* (1 차원)살창크기를 지적하는 dims행렬값넣기*/
     MPI_Dims_create(size, DIMS, dims);
/*communicator comm_cart를 가진 "고리" 위상 생성*/
  MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods,
                                        reorder, &comm_cart);
/* 매 프로쎄스는 자기의 번호를 인식한다.*/
  MPI_Comm_rank(comm_cart, &rank);
/* 매 프로쎄스는 고리를 따라 번호가 작은 방향으로 자기의 린접프로쎄스
들을 찾는다. */
    MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest);
    for (i=0; i<N; i++)
     for (j=0; j<M; j++) {
          A[i][j]=3.141528;
          B[i][j]=2.812;
          C[i][j]=0.0;
     }

/* 시작시간 */
   t1=MPI_Wtime();
/* 매 프로쎄스는 자기 대역에서 곱하기를 진행 */
/* 맨 바깥순환 for (k) 는 콤퓨터순환 */
 for (k=1; k < size; k++) {
     int d;

     d=((rank+k)%size)*N;
    /* 매 프로쎄스는 자기 대역에서의 곱하기를 진행*/
      for (j=0; j<N; j++)
         for (i1=0, j1=d; j1<d+N; i1++, j1++)
              for (i=0; i< M; i++)
                    C[j][i]+=A[j][i] * B[i][i1];
 }
```

```
    /* 매 프로세스는 낮은 번호의 린접프로세스에 B의 가로대역을 전송
한다. 즉 B의 대역을 콤퓨터고리를 따라 밀기한다. */
      MPI_Sendrecv_replace(B, EL(B), MPI_DOUBLE, dest, 12, sour, 12,
comm_cart, &st);
/* 곱하기끝시간측정*/
    t2=MPI_Wtime();
    rt=t2-t1;
    printf("rank=%d Time=%f\n", rank, rt);
/* 검사를 위하여 C의 첫행의 첫 4 개 요소들을 출력*/
    if (rank == 0) {
      for (i=0;  i < 1; i++)
       for (j=0; j < 4; j++)
        printf("C[i][j]=%f \n", C[i][j]);
   }
MPI_Comm_free(&comm_cart);
 MPI_Finalize();
 return 0;
}
```

참고문헌

[1] M.D.NcNally, Ieremy Sisssk, MPI Tutorial: University of Nortre Dame, Laboratory for Scientific Computing, 1998,
http://www.lam-mpi.org/tutorials/nd/
[2] В. Д. КОРНЕЕВ, ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В MPI; РОССИЙСКАЯ АКАДЕМИЯ НАУК, СИБИРСКОЕ ОТДЕЛЕНИЕ, ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕТАТИКИ И МАТЕМАТИЧЕСКОЙ ФИЗИКИ, НОВОСИБИРСК, 2002
[3] В. Д. КОРНЕЕВ, ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В MPI; ЯРОСЛАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П. Г. ДЕМИДОВА, ЯРОСЛАВЛЬ, 2002